НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

«На правах рукопису»
УДК 681.3

«До захисту допущено»
Завідувач кафедри
О.І. Ролік
«___»_____20__ р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології»

на тему: «Стек технологій тривимірного рендерингу в середовищі Android»

Виконав:
студент ІІ курсу, групи ІА-61м
І.З. Ашур _____

Керівник:
професор, д.ф.-м.н., професор
А.Ю. Дорошенко _____

Рецензент:
доцент, канд.фіз.-мат.наук
О.П. Ігнатенко _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2018 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – другий (магістерський) за освітньо-науковою програмою
Спеціальність – 151 «Автоматизація та комп'ютерно-інтегровані технології»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ С.Ф. Теленик
«___»_____20__ р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Ашуру Іллі Зін-Еддіновичу

1. Тема дисертації «Стек технологій тривимірного рендерингу в середовищі Android», науковий керівник дисертації Дорошенко Анатолій Юхимович, професор, д.ф.-м.н., професор, затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом дисертації _____

3. Об'єкт дослідження – тривимірний рендеринг.

4. Предмет дослідження – стек технологій для забезпечення тривимірного рендерингу в середовищі Android

5. Перелік завдань, які потрібно розробити: аналіз стеку тривимірного рендерингу Android; аналіз існуючих рішень, технік та особливостей; вибір, конфігурація та розробка стеку; визначення необхідного набору метрик; проведення профілювання; написання пояснювальної записки.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: графіки метрик ефективності імплементованого рішення; ілюстрації процесу зневадження; архітектура пропонованого рішення.

7. Перелік публікацій: «Високопродуктивний пакетний добуток матриць афінних перетворень за допомогою Android NDK та JNI» в журналі «Проблеми програмування», «Система 2D рендеринга с использованием интерфейса OpenGL ES 2.0, особенностей и техник VBO/Sprite Batch» в журналі

«Моделирование-2016», «Высокопроизводительное произведение матриц посредством Android NDK и JNI» на V Міжнародній науково-практичній конференції «Winter InfoCom Advanced Solutions 2017»

8. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Аналіз стеку тривимірного рендерингу Android	15.03.18-29.03.18	
2	Аналіз існуючих рішень, технік та особливостей	30.03.18-09.04.18	
3	Вибір, конфігурація та розробка стеку	10.04.18-17.04.18	
4	Визначення необхідного набору метрик	18.04.18-28.04.18	
5	Проведення профілювання	29.04.18-02.05.18	
6	Написання пояснювальної записки	03.05.18-14.05.18	

Студент І.З. Ашур

Науковий керівник дисертації А.Ю. Дорошенко

РЕФЕРАТ

Магістерська дисертація освітньо-кваліфікаційного рівня «магістр» на тему «Стек технологій тривимірного рендерингу в середовищі Android»: 141с., 48 рис., 22 табл., 3 додатка, 64 джерела.

Об’єкт дослідження — тривимірний рендеринг.

Мета роботи — покращення показників ефективності тривимірного рендерингу шляхом вибору, аналізу, конфігурації та імплементації стеку технологій останнього під управлінням операційної системи Android на базі графічного інтерфейсу OpenGL ES.

Тема високоефективного тривимірного рендерингу на мобільних платформах, а особливо в частині Android, є актуальною та перспективною в області систем автоматизованого проектування і розрахунку, віртуальної реальності, наукової візуалізації та відеоігор у зв’язку з постійним розвитком даного ринку, його долі та технологій.

Підмножина графічного інтерфейсу OpenGL ES (OpenGL for Embedded Systems — OpenGL для вбудованих систем) призначена для вбудованих систем — мобільних телефонів, розумних годинників, телевізорів, автомобілів, систем розумного дому, пристроїв доповненої та віртуальної реальності і т.д.

Для дослідження ефективності пропріетарних та існуючих імплементацій були використані програмні пакети GAPID та Qualcomm® Snapdragon™ Profiler.

Результати роботи впроваджені на комерційному базисі.

Дослідження має перспективи розвитку в контексті використання альтернативних, молодих графічних інтерфейсів та пошуку і імплементації нових технік та особливостей.

OPENGL ES 3.0, ANDROID, JNI, SDK, NDK, JDK, 3D, VBO, VERTEX BUFFER OBJECT, VAO, IBO, GAPID, ЕФЕКТИВНИЙ ТРИВИМІРНИЙ РЕНДЕРИНГ, JAVA, ГРАФІЧНІ ІНТЕРФЕЙСИ

ABSTRACT

Master's thesis “Android 3D rendering technology stack” consists of 141 pages, 48 figures, 22 tables, 3 appendices, and 64 sources.

The object of the study is 3D rendering.

The purpose of the work is to enhance 3D rendering performance via choosing, analyzing, configuring and implementing high-performance 3D rendering stack for Android OS with OpenGL ES interface.

High-performance 3D rendering on mobile platforms is relevant and promising topic in the field of automated design and calculation systems, virtual reality, scientific visualization and video games because of constant development of market share and technology level.

OpenGL for Embedded Systems (OpenGL ES or GLES) is designed for embedded systems like smartphones, tablet computers, video game consoles and PDAs.

GAPID and Qualcomm® Snapdragon™ Profiler software suits were used to investigate and analyze performance of developed and existing implementations.

Development results are used on commercial basis.

The research has prospects for development in context of the use of alternative, high-end graphic interfaces and the search and implementation of new techniques and features in scope of current implementation.

OPENGL ES 3.0, ANDROID, JNI, SDK, NDK, JDK, 3D, VBO, VERTEX BUFFER OBJECT, VAO, IBO, GAPID, HIGH-PERFORMANCE 3D RENDERING, JAVA, GRAPHICAL API

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

- OpenGL ES — від англ. Open Graphics Library for Embedded Systems
- JVM — від англ. Java Virtual Machine
- JDK — від англ. Java Development Kit
- GPU — від англ. Graphics Processing Unit
- AOSP — від англ. Android Open Source Project
- OS — від англ. Operating System
- HAL — від англ. Hardware abstraction layer
- GHz — від англ. Gigahertz, гігагерц
- ns — від англ. Nanosecond, наносекунда
- API — від англ. Application Programming Interface
- SDK — від англ. Software Development Kit
- NDK — від англ. Native Development Kit
- GLSL — від англ. OpenGL Shading Language
- byte, short, float, double, vec2, vec3, vec4, ivec3, dvec4, int, uint — типи даних в C
- VBO — від англ. Vertex Buffer Object
- VAO — від англ. Vertex Array Object
- IBO — від англ. Index Buffer Object
- JNI — від англ. Java Native Interface
- DSP — від англ. Digital signal processing

ВСТУП

Тема високоефективного тривимірного рендерингу на мобільних платформах, а особливо в частині Android, є актуальною та перспективною в області систем автоматизованого проектування і розрахунку, віртуальної реальності, наукової візуалізації та відеоігор у зв’язку з постійним розвитком даного ринку, його долі та технологій.

Мета магістерської дисертації — покращення показників ефективності тривимірного рендерингу шляхом вибору, аналізу, конфігурації та імплементації стеку технологій останнього під управлінням операційної системи Android на базі графічного інтерфейсу OpenGL ES.

Об’єкт дослідження — тривимірний рендеринг.

Предмет дослідження — стек технологій для забезпечення тривимірного рендерингу в середовищі Android. [3]

Результати даного дослідження вдосконалюють існуючі підходи високоефективного тривимірного рендерингу, мають перспективи подальшого розвитку та успішно впроваджені на комерційній основі.

За результатами досліджень опубліковані статті та тези: «Високопродуктивний пакетний добуток матриць афінних перетворень за допомогою Android NDK та JNI» в журналі «Проблеми програмування» (додаток А), «Система 2D рендеринга с использованием интерфейса OpenGL ES 2.0, особенностей и техник VBO/Sprite Batch» в журналі «Моделирование-2016» (додаток Б), «Высокопроизводительное произведение матриц посредством Android NDK и JNI» на V Міжнародній науково-практичній конференції «Winter InfoCom Advanced Solutions 2017» (додаток В)

РОЗДІЛ 1. ДЕТАЛЬНИЙ ОГЛЯД СТЕКУ ТЕХНОЛОГІЙ ТРИВИМІРНОГО РЕНДЕРИНГУ В СЕРЕДОВИЩІ ANDROID

У даному розділі наведений огляд доступних на момент дослідження методів рендерингу тривимірної графіки під управлінням цільової платформи, а саме реалізації програмного і апаратно-прискореного рендерингу.

1.1 Програмний рендеринг

Програмний рендеринг [4] представляє процес побудови зображення без залучення ресурсів графічного процесору [5]. Такий рендеринг може відбуватися як в поточному часі при відображенні великої кількості кадрів за відносно короткий проміжок часу, так і у відкладеному режимі, де нема строгих лімітів на витрати часу для відображення одного кадра. В контексті даної роботи актуальні лише засоби спрямовані на рендеринг в живому часі.

Сучасні центральні процесори, за рахунок яких може здійснюватися програмний рендеринг, не можуть конкурувати з ефективністю та потужністю графічних процесорів, створених безпосередньо для задач рендерингу.

У контексті сучасних версій операційної системи Android програмний рендеринг здебільшого не використовується.

1.2 Апаратно-прискорений рендеринг

В області комп’ютеризації під апаратним прискорення розуміють використання апаратного забезпечення для виконання деяких функцій швидше у порівнянні з виконанням програм процесором загального призначення. Прикладами апаратного прискорення може бути блокове прискорення виконання у графічному процесорі та інструкції комплексних операцій у центральному процесорі. [6]

Зазвичай процесори виконують роботу послідовно, а інструкції — по черзі. Для покращення продуктивності застосовуються різноманітні способи, апаратне прискорення — один із них. Основна відмінність апаратних засобів від програмних полягає у паралелізації, що дозволяє апаратному забезпеченню виконувати операції набагато швидше, ніж програмному. Апаратні прискорювачі спеціально спроектовані для програмного коду, що створює велике обчислювальне навантаження. В залежності від ступені деталізації, апаратне прискорення може варіюватись від невеликої функціональної одиниці до великого функціонального блоку.

Апаратне забезпечення, що виконує прискорення в вигляді окремої одиниці центрального процесору, називається апаратним прискорювачом, але частіше визначається як графічний прискорювач або прискорювач роботи з плаваючою комою. На рисунку 1.1 зображена загальна схема використання апаратних і програмних ресурсів апаратно-прискореного рендерингу в частині пам’яті та процесорів. [7]

Рисунок 1.1

1.3 Стек технологій

1.3.1 Android

Android — це мобільна операційна система, розроблена компанією Google, на основі модифікованої версії ядра Linux та іншого програмного забезпечення з відкритим початковим кодом, розроблена переважно для сенсорних мобільних пристроїв, таких як смартфони та планшети. Крім того, компанія Google розробила Android TV для телевізорів, Android Auto для автомобілів та Wear OS для наручних годинників, кожна з яких має спеціалізований інтерфейс користувача. Варіанти Android також використовуються на ігрових консолях, цифрових камерах, ПК та в іншій електроніці. [8]

Спочатку розроблений компанією Android Inc., яку Google придбав у 2005 році, Android був представлений в 2007 році, а перший комерційний пристрій Android запущений в продаж у вересні 2008 року. Операційна система з тих пір пройшла кілька основних релізів, з поточною версією 8.1 «Oreo» , випущеною в грудні 2017 року. Основний початковий код Android відомий як проект з відкритим кодом Android (AOSP), і в першу чергу ліцензується в рамках ліцензії Apache.

Android є найпопулярнішою операційною системою по всьому світу на смартфонах з 2011 року та планшетах з 2013 року. З травня 2017 року вона має понад 2 мільярди активних користувачів, що є найбільшою встановленою базою будь-якої операційної системи, а з 2017 року — Google Play налічує понад 3,5 мільйони додатків. [9]

На наступному рисунку зображена доля різноманітних версій даної операційної системи поміж існуючих пристроїв станом на 7 травня 2018 року [10] (рисунок 1.2):

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.5%
4.2.x		17	2.2%
4.3		18	0.6%
4.4	KitKat	19	10.3%
5.0	Lollipop	21	4.8%
5.1		22	17.6%
6.0	Marshmallow	23	25.5%
7.0	Nougat	24	22.9%
7.1		25	8.2%
8.0	Oreo	26	4.9%
8.1		27	0.8%

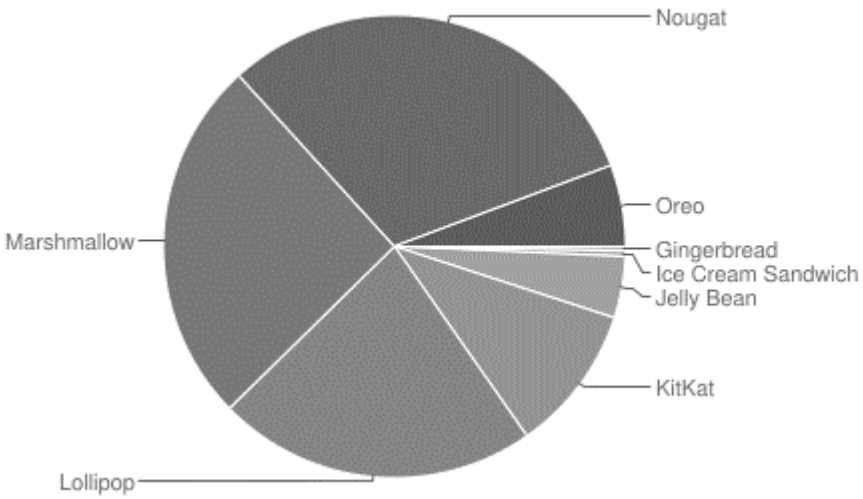


Рисунок 1.2 — Доля різноманітних версій Android поміж існуючих пристроїв станом на 7 травня 2018 року

1.3.2 Стек рендерингу Android

В контексті огляду методів рендерингу необхідно також оглянути графічну підсистему платформи Android. На рисунках 1.3 і 1.4 представлені відношення між різними графічними компонентами і середовищем платформи ОС Android. [11]

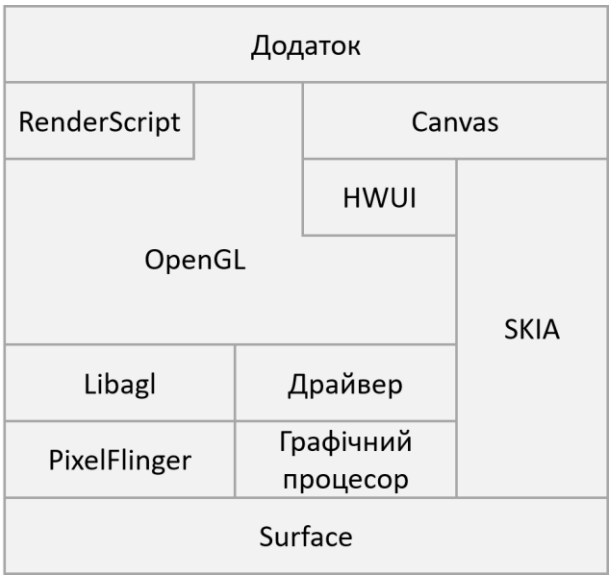


Рисунок 1.3 — Відношення між різними графічними компонентами

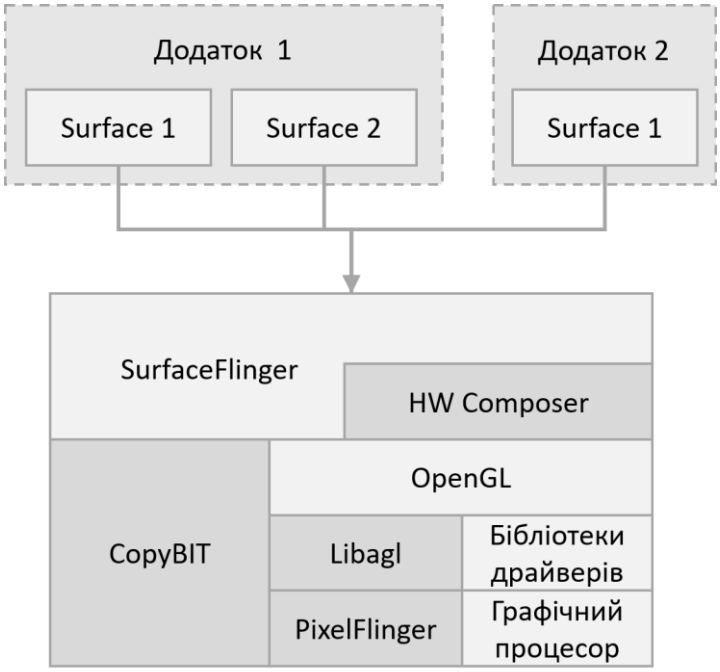


Рисунок 1.4 — Графічний стек платформи Android

На рисунках зображені наступні компоненти:

1. Графічний процесор (GPU) — спеціалізований апаратний рушій для прискорення графічних операцій. Головна відмінність від центрального процесора полягає у тому, шо графічний процесор заточений під виконання паралельних задач, що необхідно для більшості графічних операцій.

В перших версіях мобільних приладів під управлінням Android ОС, графічні процесори були опціональними, але з плином часу присутність останнього стала обов'язковою. Системи без графічних процесорів використовують програмний стек OpenGL ES, котрий складається з Libagl та PixelFlinger, іноді з апаратною підтримкою CopyBIT.

Програмна емуляція OpenGL на Android не підтримує стандарт OpenGL ES 3.0 та вище, котрий сьогодні використовується багатьма ключовими частинами операційної системи, такими як HWUI, SurfaceTextures. Більш того, сучасні мобільні прилади отримують постійно зростаючу роздільну здатність, за рахунок чого програмні методи рендерингу не можуть забезпечувати прийнятних значень Fillrate, а, відповідно, не можуть забезпечувати необхідний досвід взаємодії. На рисунках 1.5 та 1.6 представлена статистика розповсюдження роздільної здатності та розмірів екранів, їх загальна спільна відповідність. [12]

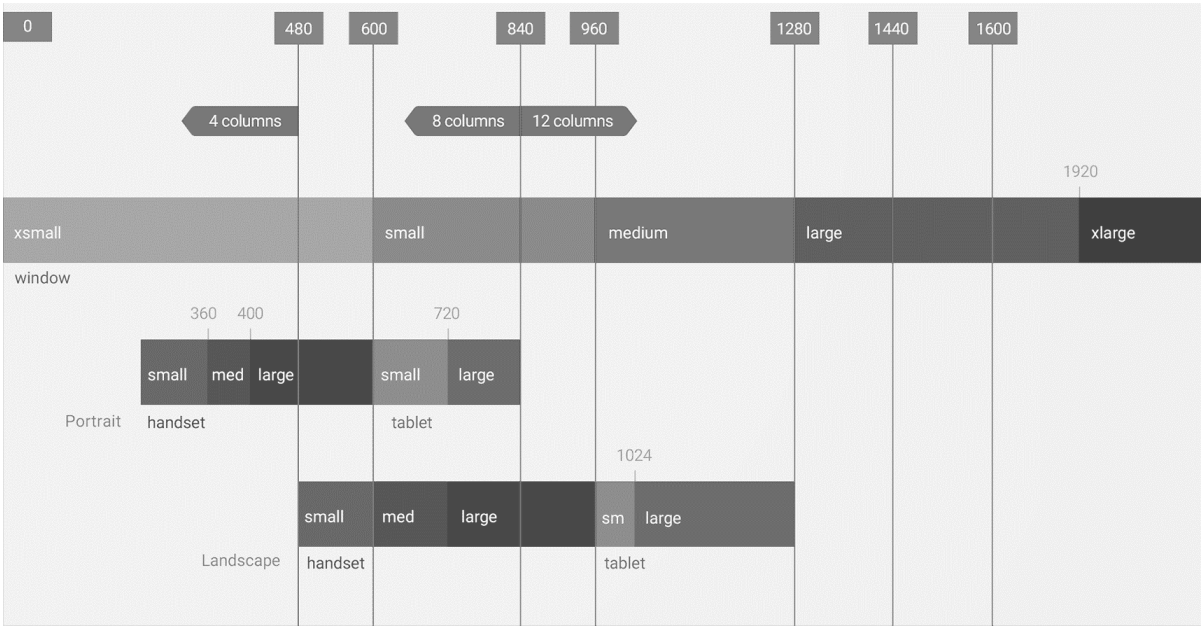


Рисунок 1.5

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	0.4%					0.1%	0.5%
Normal		0.9%	0.3%	27.3%	39.3%	23.3%	91.1%
Large		2.4%	1.5%	0.4%	0.7%	0.5%	5.5%
Xlarge		1.8%		0.6%	0.5%		2.9%
Total	0.4%	5.1%	1.8%	28.3%	40.5%	23.9%	

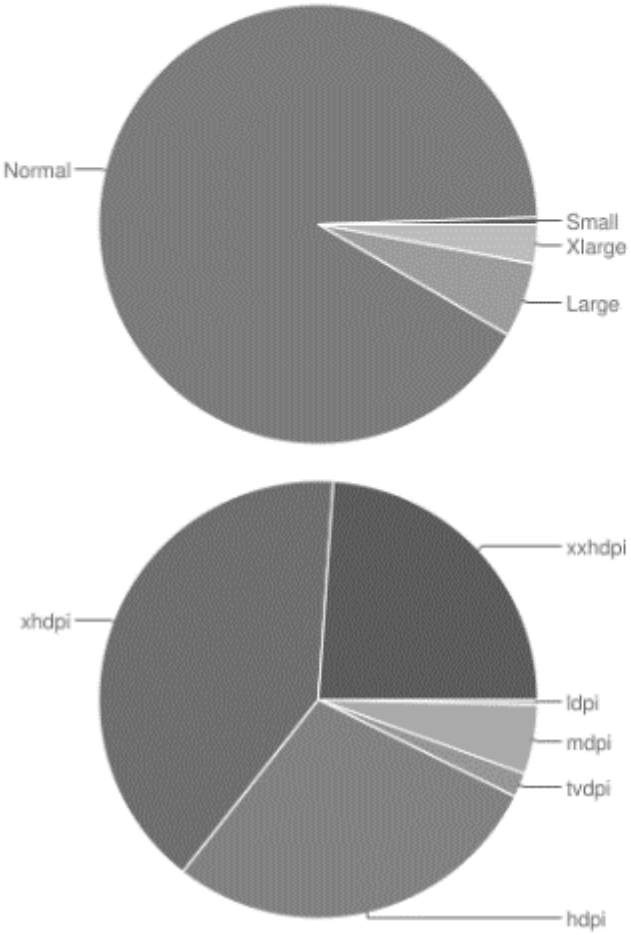


Рисунок 1.6

Дані з даних рисунків зібрані протягом тижневого вікна станом на 7 травня 2018 року.

Fillrate — швидкість заповнення пікселями, одна з найважливіших характеристик графічного процесора. Цей параметр представляє число пікселів, для котрих графічний процесор може прорахувати всі необхідні операції за одиницю часу. Чим більша швидкість заповнення, тим краще. На сучасних графічних процесорах цей параметр лінійно залежить від кількості піксельних конвеєрів. Fillrate часто буває вузьким місцем продуктивності рендерингу. Це часто означає, що задані занадто важкі в обчисленні піксельні шейдери або занадто великий обсяг даних для рендерингу в контексті даної апаратної конфігурації та використаних особливостей, в наступних розділах вузькі місця продуктивності будуть оглянуті детальніше. [13]

2. Canvas. Вбудований Android клас для відображення графіки. Використовує Skia та HWUI (зараз — за замовчуванням)

3. Skia. Інтерфейс програмування додатків (далі — Application programming interface, API) для повністю програмного двовимірного рендерингу. Зараз в більшості замінений на HWUI з міркувань продуктивності. [14]

4. HWUI. Вбудована бібліотека для апаратно-прискореного рендерингу елементів графічного інтерфейсу. Була вперше представлена у версії Honeycomb (API Level 11) для забезпечення необхідного досвіду взаємодії в частині відгучливості, швидкості і плавності графічного інтерфейсу. Для роботи HWUI необхідний графічний процесор сумісний зі стандартом OpenGL ES 2.0, котрий, як було вказано вище, не може емулюватися програмними методами. [15]

5. RenderScript. Програмний інтерфейс та компонент Android, не цікавий в рамках даного дослідження. [16]

6. Surface. Компонент Android. Представляє кадровий буфер, в котрий додатки відображують графічний контент. В якості додатку може виступати як будь-який додаток, що використовує OpenGL для безпосереднього рендерингу в Surface, так і простий додаток, що відображає віджети, текст, зображення та інші візуальні компоненти. [17]

7. SurfaceFlinger. Display server, Window server для Android — програмний прошарок для координації вводу-виводу підпорядкованих клієнтів, операційної

системи та апаратного забезпечення. Він визначає джерело відображеного на екрані контенту і принципи його накладення. [18]

8. HW Composer. Прошарок апаратних абстракцій (Далі — Hardware Abstraction Layer, HAL). Використовується у SurfaceFlinger для ефективної комбінації ресурсів апаратного забезпечення.

9. CopyBit. Прошарок апаратних абстракцій. На даний момент не підтримується, тому його огляд опускається.

10. Libagl/PixelFlinger. Libagl — бібліотека для програмної емуляції інтерфейсів OpenGL ES версії 1.0 та 1.1. Використовує PixelFlinger для реалізації команд OpenGL.

1.3.3 OpenGL

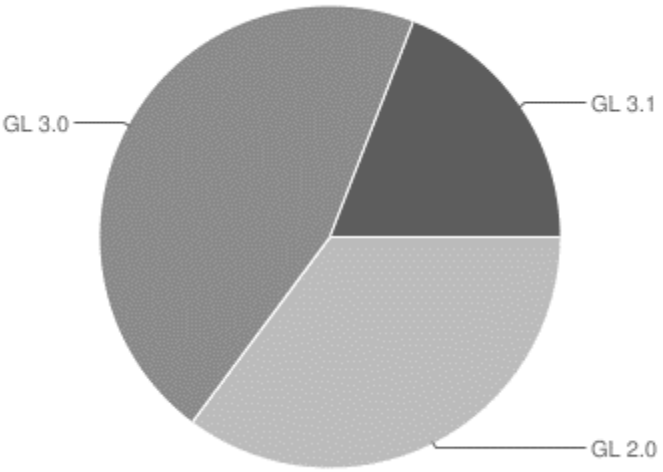
Open Graphics Library (OpenGL) — крос-мовний, багатоплатформовий прикладний програмний інтерфейс (API) для рендерингу двовимірної та тривимірної векторної графіки. Даний програмний інтерфейс зазвичай використовується для взаємодії з графічним процесором (GPU) для досягнення апаратно-прискореного рендерингу.

Silicon Graphics Inc., (SGI) почала розробку OpenGL у 1991 році і випустила перший реліз у січні 1992. Додатки в області систем автоматизованого проектування і розрахунку, віртуальної реальності, наукової візуалізації, симуляції польотів та відеоігор широко використовують OpenGL. Починаючи з 2006 року специфікацію OpenGL контролює некомерційний технологічний консорціум Khronos Group. [19]

1.3.3.1 OpenGL ES 3.0

OpenGL for Embedded Systems (OpenGL для вбудованих систем) — підмножина інтерфейсу OpenGL для рендерингу двовимірної та тривимірної векторної графіки. Вона спеціально спроектована для вбудованих систем по

типу смартфонів, планшетів, ігрових консолей, кишенькових персональних комп’ютерів. OpenGL ES — найрозповсюдженіший інтерфейс тривимірної графіки в історії. На рисунку 1.7 зображена статистика розповсюдження версій даного інтерфейсу на приладах з операційною системою Android станом на 7 травня 2018 року. [20, 21]



OpenGL ES Version	Distribution
2.0	35.2%
3.0	45.7%
3.1	19.1%

Рисунок 1.7 — Розповсюдження версій OpenGL ES на приладах з операційною системою Android станом на 7 травня 2018 року.

1.3.4 Vulkan

Vulkan — багатоплатформне API для 3D графіки і супроводжуючих обчислень, представлене компанією Khronos Group. Початково розробка даного API була в рамках ініціативи OpenGL наступного покоління і на деяких презентаціях проект був анонсований під назвою «glNext», який був покликаний

вирішити існуючі проблеми та недоліки OpenGL, згодом цей проект отримав саме назву Vulkan. Враховуючи основні принципи, використані в розробці Vulkan, — його застосування має принести перевагу в швидкодії в порівнянні з OpenGL, шляхом ефективнішого використання GPU. Деякі компоненти Vulkan були позичені з іншого API — Mantle від компанії AMD, який свого часу також створювався для заміни вже існуючих DirectX і OpenGL. [22]

Vulkan підтримується на Android починаючи з версії API level 24.

1.4 Огляд існуючих рішень

1.4.1 libGDX

libGDX — це Java фреймворк, який надає крос-платформне API для розробки ігор і додатків, що працюють в режимі реального часу. Це високопродуктивний, кросплатформний ігровий фреймворк, що в першу чергу використовується для написання ігрових рушіїв та ігор. Позиціонується як фреймворк та дозволяє максимально зосередитися на міцному фундаменті, замість того, щоб намагатися реалізувати найновіше і найкраще з ігрових рушіїв. libGDX надає гнучкість і дозволяє уникнути суворої методології. За допомогою даної бібліотеки, можна використовувати один і той же код як для систем настільних комп'ютерів так і мобільних систем. Бібліотека є кросплатформенною і підтримує Windows, Linux, Mac OS X, Android, iOS, та браузери з підтримкою WebGL. [23]

1.5 Огляд технік, особливостей і методів тривимірного рендерингу

Для досягнення максимальної ефективності тривимірного рендерингу, необхідно орієнтуватись на графічний стек, що є максимально наближеним до взаємодії з графічним процесором через мінімальну кількість прошарків-посередників, що в результаті забезпечить мінімальні накладні затрати ресурсів

і максимальну гнучкість можливої імплементації. В якості такого стека виступає ціпка Додаток — Рушій рендерингу — OpenGL ES API — Драйвер графічного процесору — Графічний процесор — Surface (рисунок 1.9). В якості доказу непридатності методів-посередників для задач високоефективного тривимірного рендерингу приведемо сторонні тести продуктивності даних методів у порівнянні з базовим рендерингом за використанням OpenGL ES (рисунок 1.8).

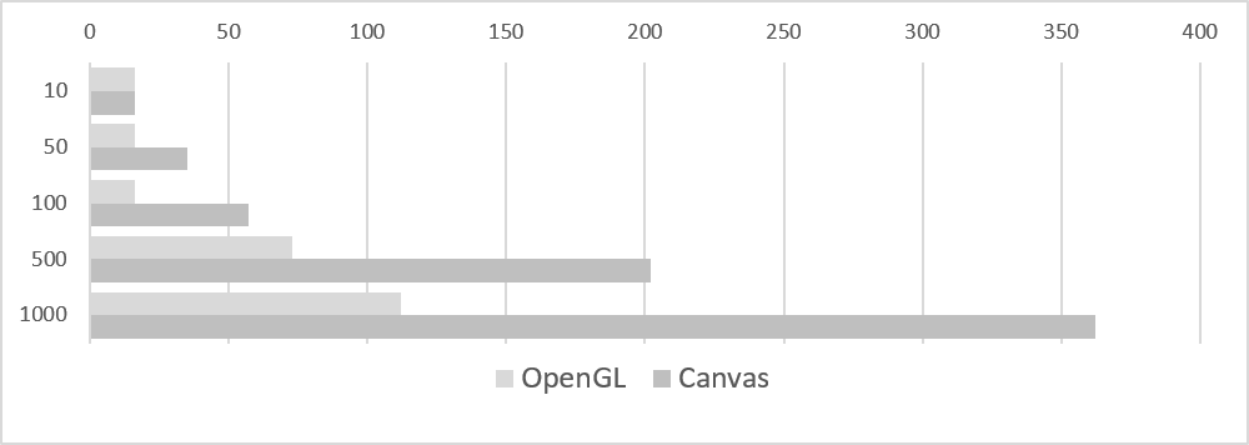


Рисунок 1.8 — Узагальнений тест продуктивності рендерингу за використанням OpenGL ES та Canvas. Кількість примітивів по вертикалі, час рендерингу одного кадру в мілісекундах — по горизонталі. Використане апаратне забезпечення: Quad-core 1.2 GHz Cortex-A7, Adreno 305 [24]

Даний тест не вимагає детального огляду та базується на двовимірному рендерингу з використанням мінімуму необхідних особливостей, мінімуму оптимізації та простих шейдерів з ціллю показати порядок різниці на базовому рівні взаємодії зі стеками рендерингу.

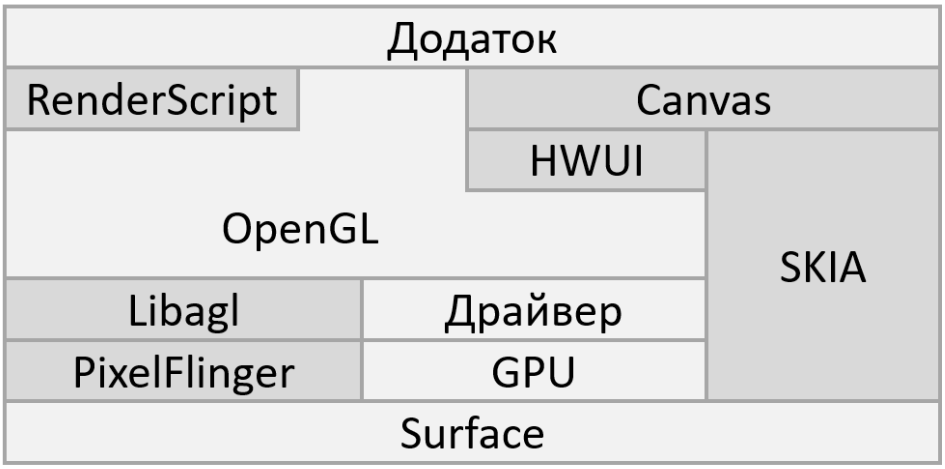


Рисунок 1.9 — Обраний стек позначений світлим кольором

1.5.1 Методи рендерингу

Ключова особливість дослідження — високопродуктивна імплементація та використання обраного стеку тривимірного рендерингу, а насамперед — взаємодія з OpenGL ES API. Android підтримує високопродуктивний рендеринг двовимірної та тривимірної графіки за допомогою Open Graphics Library (далі — OpenGL), а саме — OpenGL ES API. Android підтримує цілий ряд версій OpenGL ES API:

- OpenGL ES 1.0 и 1.1 підтримується в Android версії 1.0 та вище
- OpenGL ES 2.0 — Підтримується в Android версії 2.2 (API Level 8) та вище
- OpenGL ES 3.0 — Підтримується в Android версії 4.3 (API Level 18) та вище
- OpenGL ES 3.1 — Підтримується в Android версії 5.0 (API Level 21) та вище

Підтримка OpenGL ES 3.0 API та вище зобов’язує виробника мобільного пристрою власноруч реалізовувати вказану специфікацію. Таким чином, пристрій під керуванням Android 4.3 або вище може не підтримувати відповідну версію інтерфейсу.

Android підтримує OpenGL за допомогою власного пакету розробки програмного забезпечення, як на базі Java/Kotlin, так і на базі нативного пакету розробки (Далі — Native Development Kit, NDK). [25]

1.5.1.1 Конвеєр OpenGL ES 3.0

На рисунках 1.10 та 1.11 представлені схеми конвеєрів OpenGL: конвеєра Fixed function та програмованого конвеєра. Програмований конвеєр був введений з появою стандарту OpenGL ES 2.0. Програмовані етапи конвеєра позначені світлим. Далі будуть приведені його особливості у порівнянні з конвеєром Fixed function:

- OpenGL ES Shading Language — додає підтримку мови високого рівня для програмування шейдерів, адаптованого для вбудованих систем.
- Програмований OpenGL ES 2.0+ конвеєр заміщає Fixed function конвеєр OpenGL 1.x. (Як це показано на рисунках нижче).
- Використання шейдерів для зменшення складності і збільшення продуктивності програмованих графічних підсистем.
- Використання Frame Buffer Objects для спрощення управління закадрового рендерингу, включаючи обробку в текстуру.

OpenGL ES версій 1.0, 1.1, 2.0, 3.0 та 3.1 забезпечують високопродуктивні інтерфейси рендеринга графіки. Програмування під OpenGL ES API версій 2.0 та 3.0 / 3.1 багато в чому схоже. Програмування під OpenGL ES API версій 1.0 / 1.1 і 2.0 / 3.0 / 3.1 сильно відрізняється. Виходячи з усіх представлених факторів, розробнику необхідно ретельно підходити до вибору потрібної версії інтерфейсу OpenGL ES. Можливо виділити кілька основних факторів вибору:

- Продуктивність. У загальному випадку, OpenGL ES версій 2.0 та 3.0 / 3.1 забезпечують кращу продуктивність рендерингу в порівнянні з інтерфейсами версій 1.0 / 1.1. Порівняльні показники продуктивності можуть відрізняться в залежності від реалізації інтерфейсу виробником, але така можливість практично скасовується у зв'язку з масовим переходом на OpenGL ES пізніх версій. Докладні дані по статистиці поширення версій були приведені в розділі 1.

- Сумісність з пристроями. Необхідно зважувати актуальні дані по статистиці поширення версій Android і підтримуваних версій OpenGL ES з метою забезпечення максимальної сумісності з більшою частиною пристроїв на ринку.
- Гнучкість розробки. OpenGL ES версій 1.0 / 1.1 надають застарілий Fixed function конвеєр, який більше не підтримується в версіях 2.0 і 3.0 / 3.1. Переваги Fixed function конвеєра — низький поріг входження і простота реалізації базових задач. Переваги програмованого конвеєра — гнучкість, а, як наслідок — потенційна продуктивність рішень.
- Управління графікою. OpenGL ES версій 2.0 та 3.0 / 3.1 надають великі можливості управління рендерингом завдяки повністю програмованим шейдерам конвеєру OpenGL. Використовуючи більш прямий контроль графічного конвеєра, розробник здатний реалізовувати завдання часто нездійсненні за допомогою OpenGL ES версій 1.0 / 1.1.

На рисунках 1.10 і 1.11 зображені загальні схеми конвеєрів та їх програмованих і непрограмованих частин.

Таким чином, рішення про вибір необхідної версії інтерфейсу має прийматися з урахуванням наведених факторів і в цілях забезпечення максимального досвіду взаємодії. [26]

Для даного дослідження прийнято рішення використовувати програмований конвеєр OpenGL ES версії 3.0 з міркувань оптимальних показників гнучкості, функціональності, продуктивності і сумісності.

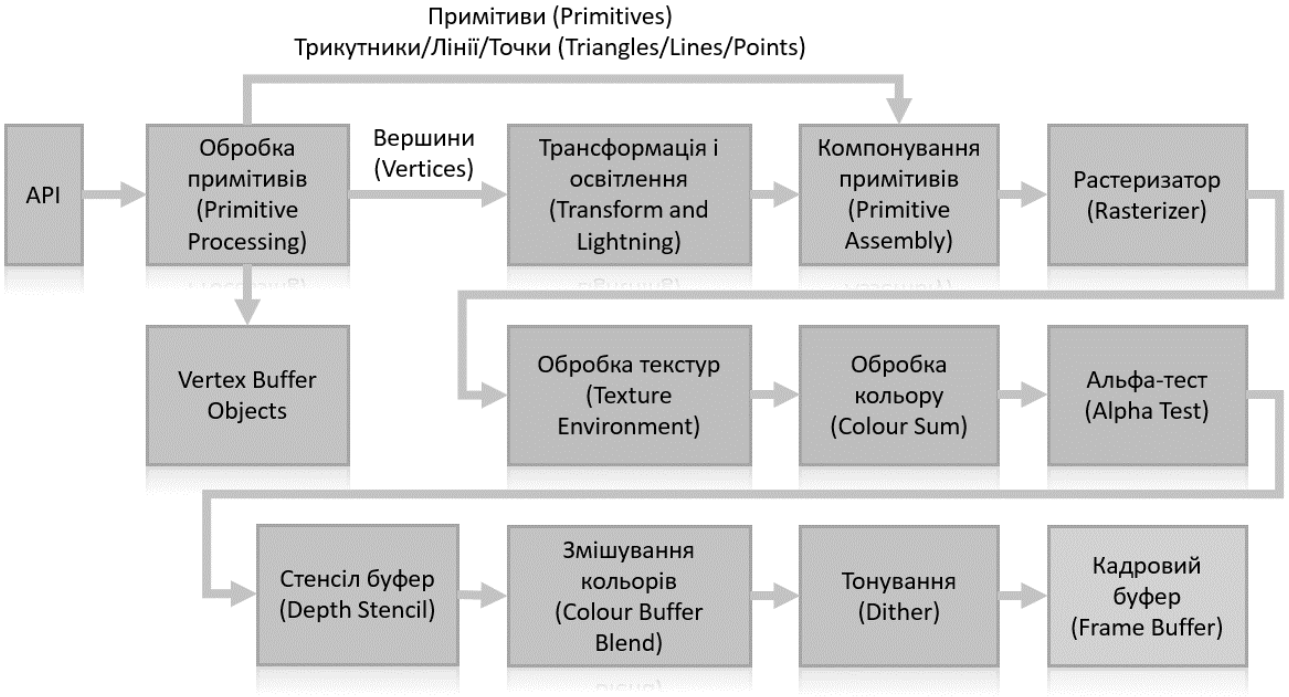


Рисунок 1.10 — Fixed function конвеєр

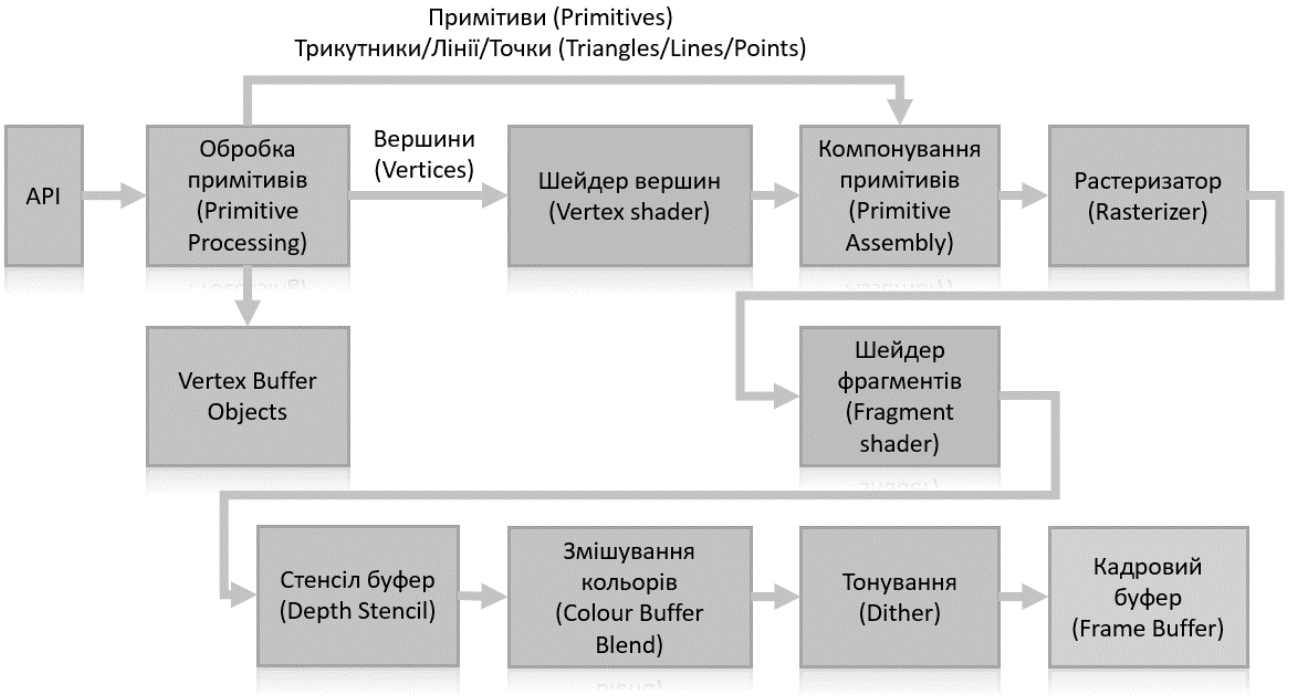


Рисунок 1.11 — Програмований конвеєр

1.5.1.2 Паралелізм OpenGL ES 3.0

Графічні процесори високопаралельні. Це є головною причиною їх високої продуктивності. Вони реалізують два види паралелізму: вертикальний і горизонтальний.

Конвеєр OpenGL ES 3.0 / 3.1 виконується в наступному порядку, зображеному на рисунку 1.12:

Рисунок 1.12

- Вертикальний паралелізм описує паралельну обробку на різних стадіях конвеєра. В контексті обробки даних графічним процесором, прості завдання відповідають простішим блокам обробки даних, що покращує показники енергоефективності та продуктивності.
- Горизонтальний паралелізм описує можливість обробки завдань в ряді конвеєрів. Це дозволяє досягти ще більшого паралелізму, ніж у випадку з вертикальним паралелізмом в єдиному конвеєрі. В контексті графічного процесора, горизонтальний паралелізм — важлива особливість, завдяки якій досягається висока продуктивність на сучасних графічних процесорах.

1.5.1.3 Стадії конвеєра OpenGL ES 3.0

- Визначення вершин (Vertex Data / Vertex Buffer) — стадія передбачає створення набору вершин в пам'яті додатку. За допомогою цих вершин згодом будуть складені графічні примітиви, а з примітивів — графічні елементи.
- Шейдер вершин (Vertex Shader). Вміст набору даних надходить на обробку в програму — вершинний шейдер. Шейдер здійснює операції над вершинами — наприклад, застосовує матриці перетворень. Даний етап є програмованим, розробник здатний реалізовувати власну логіку обробки.
- Компонування примітивів (Primitive Assembly) — на даному етапі конвеєр перетворює результати роботи вершинного шейдера в примітиви — точки, лінії, трикутники. Примітиви будуть детально розглянуті в наступних підрозділах. На цьому ж етапі визначається, чи входить той чи інший примітив в видимий простір, за потреби — проводиться обрізка і передача примітивів на наступний етап.
- Растеризація (Rasterization) — на даному етапі отримані примітиви перетворюються в фрагменти. Фрагмент являє собою набір значень, що

відповідає за частину растрового примітиву. Розмір фрагмента визначається екранним пікселем, станом конвеєра, параметрами згладжування. Хоча б один фрагмент буде згенеровано для кожного пікселя, зайнятого растровим примітивом.

- Фрагментний шейдер (Fragment Shader) — програмована стадія конвеєра. Даний шейдер здійснює обробку кожного отриманого з попереднього етапу фрагмента. Фрагментний шейдер здійснює перетворення зі складовою примітивів, що відповідає за колір.
- Фрагментні операції (Per-Fragment Operations). Це — останній етап перетворення фрагментів в пікселі кадрового буфера. Включає в себе генерацію текселей (мінімальна одиниця текстури) і ряд таких операцій, як: перевірка на входження фрагмента в прямокутник відтинання, згладжування, застосування трафаретів, тест буфера глибини, змішування кольорів, тонування.

1.5.2 Основні поняття в конвеєрі OpenGL ES 3.0

В даному розділі здійснений огляд нативного інтерфейсу OpenGL API

1.5.2.1 Процес створення об’єктів рендерингу

Vertex Specification — процес створення і конфігурації об'єктів, необхідних для рендеринга за допомогою певної програми-шейдера.

Використання вершинних даних в конвеєрі для рендеринга передбачає створення потоку вершин і конфігурації машини OpenGL з метою визначення принципів взаємодії з даним потоком.

Для того, щоб отримати можливість рендерингу в принципі, необхідно, як це було зазначено вище, використовувати шейдер конвеєра, зокрема — вершинний шейдер. Визначені розробником вхідні змінні утворюють список

очікуваних вершинних атрибутів для даного шейдера. Цей набір атрибутів визначає значення, які надає потік вершинних даних для забезпечення рендеринга за допомогою даного шейдера.

Для кожного атрибута в шейдері розробник зобов'язаний надати масив даних, відповідних атрибуту. Кожен такий масив повинен містити однакову кількість елементів. Слід зазначити, що використовуються тут масиви на порядок більш гнучкі, ніж Сі-подібні, але загальні принципи функціонування зберігаються.

Порядок вершин в потоці також є важливим фактором. Цей порядок визначає те, як OpenGL буде обробляти і рендерити примітиви, надані потоком. Безпосередньо самі примітиви будуть розглянуті в наступних підрозділах. Існує два способи рендеринга даного масиву вершин. Перший — генерація потоку в тому ж порядку, в якому впорядкований масив вершин. Другий — використання індексних списків для визначення порядку вершин в потоці. Індексний список визначає порядок одержуваних вершин і може багаторазово вказувати на один і той же елемент в масиві вершин.

В якості прикладу наведемо наступний масив тривимірних координат:

$$\{\{1, 1, 1\}, \{0, 0, 0\}, \{0, 0, 1\}\} \quad (1.1)$$

При використанні даного потоку конвеєр OpenGL отримає і обробить три вершини в заданому порядку (зліва направо). Проте, існує можливість визначити список індексів, за допомогою яких будуть визначені використовувані вершини і їх порядок.

В якості прикладу наведемо наступний список індексів:

$$\{2, 1, 0, 2, 1, 2\} \quad (1.2)$$

Під час рендерингу наведеного вище масиву координат з даним списком індексів отримаємо наступний потік вершин:

$$\{\{0, 0, 1\}, \{0, 0, 0\}, \{1, 1, 1\}, \{0, 0, 1\}, \{0, 0, 0\}, \{0, 0, 1\}\} \quad (1.3)$$

Індексний список — спосіб перевизначення вершинного масиву без фактичного його зміни. Найчастіше, ця особливість може бути застосована з метою компресії, і, як наслідок, зменшення витрат ресурсів на передачу даних по конвеєру, так як складні геометрії можуть мати багато повторюваних вершин. Більш того, дана особливість дозволяє одноразово створити і запам'ятати масив індексів в рамках стану конвеєра. Використання такого масиву також більш ефективно в порівнянні з повторним визначенням вершин у зв'язку з тим, що вершина може займати на порядок більше пам'яті у порівнянні з індексом, який часто займає від 2 до 4 байт. Детальніше індексні буфери та можливості компресії будуть оглянуті в наступних розділах.

Потік вершинних даних також може містити безліч атрибутів. Наприклад, наведений на початку масив вершин можна доповнити масивом текстурних координат для кожної вершини 1.4 і отримати потік такого вигляду 1.5:

$$\{\{0, 0\}, \{0.5, 0\}, \{0, 1\}\} \quad (1.4)$$

$$\{[\{0, 0, 1\}, \{0, 1\}], [\{0, 0, 0\}, \{0.5, 0\}], \ldots, [\{0, 0, 1\}, \{0, 1\}]\} \quad (1.5)$$

Слід пам'ятати, що OpenGL не дає можливості використовувати різні масиви індексів для одного і того ж масиву атрибутів. [27]

1.5.2.2 Примітиви

Поняття примітиву [28] в OpenGL використовується для опису двох схожих концепцій. Перша з них є механізмом конвеєра, що відповідає за визначення представлення потоку вершин під час рендерингу. Такі потоки вершин можуть бути довільної довжини.

Друге значення примітиву є результатом інтерпретації потоку вершин як частини етапу компонування примітивів, розглянутого в попередніх розділах. Таким чином, обробка потоку вершин дає на виході відповідну послідовність примітивів.

На рисунку 1.13 приведено схематичне зображення основних примітивів і їх типів. Детальніше примітиви будуть розглянуті нижче.

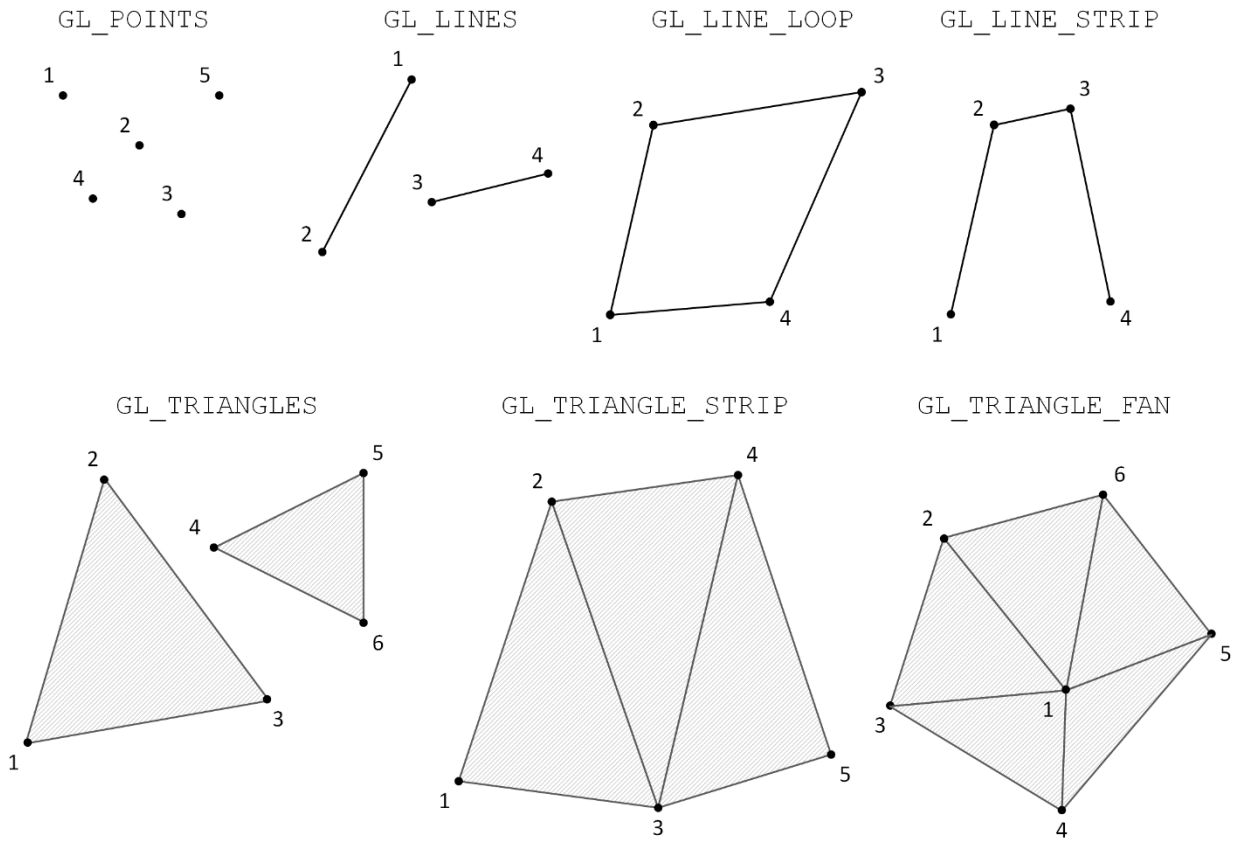


Рисунок 1.13

1.5.2.2.1 Точки

Існує єдиний тип примітивів-точок в поняттях OpenGL, іменований GL_POINTS. Даний спосіб інтерпретації потоку вершин дозволяє представляти кожну окрему вершину в потоці як точку. Точки з розміченою на них текстурою часто називають точковими спрайтами.

Точки растеризуються як вирівняні по екрану квадрати певного розміру. Розмір може бути визначений за допомогою викликів функції API glEnable з одним з параметрів виду GL_PROGRAM_POINT_SIZE.

Розмір визначає кількість віконних пікселів на сторону представленого квадрата. Позиція точки відповідає центру квадрата.

Встановлений розмір повинен бути більше 0, в іншому випадку можливе виникнення невизначеної поведінки. Доступні рамки розмірів визначаються конкретною реалізацією API і повертаються за допомогою параметрів виду GL_POINT_SIZE_RANGE / GRANULARITY.

1.5.2.2.2 Лінії

Існують три типи примітивів-ліній, основаних на різних інтерпретаціях потоку вершин.

- GL_LINES — Кожна наступна пара вершин інтерпретуються як лінія. При виникненні непарної кількості вершин в потоці — додаткова вершина ігнорується.
- GL_LINE_STRIP — Суміжні вершини формують лінії. Таким чином, маючи n вершин в потоці на виході отримуємо n-1 ліній. При існуванні лише однієї вершини в потоці команда ігнорується.
- GL_LINE_LOOP — Тип, подібний попередньому за тим винятком, що перша і остання вершина також інтерпретуються як лінія. При існуванні лише однієї вершини в потоці команда ігнорується. Лінія між першою і останньою вершиною обробляється останньою.

1.5.2.2.3 Трикутники

Трикутний примітив формується за допомогою трьох вершин. Він являє собою двовимірну фігуру з мінімально можливою кількістю вершин. З цієї

причини більшість реалізацій конвеєрів і високорівневих систем спроектовані для роботи з даним примітивом. Зрозуміло, що даний примітив може бути виключно планарним.

Існує три типи даного примітиву, основаних на різних інтерпретаціях потоку вершин.

- GL_TRIANGLES — Кожна наступна група з трьох вершин інтерпретується як трикутник.
- GL_TRIANGLE_STRIP — Кожна група з трьох вершин інтерпретується як трикутник. Потік довжиною в n вершин дасть на виході n-2 трикутних примітивів даного типу.
- GL_TRIANGLE_FAN — Перша вершина в потоці фіксується. З цього моменту, кожна наступна група з двох вершин формує трикутник з раніше зафіксованої. Таким чином, можна подати такі індекси утворених трикутників для вершинного потоку на 5 елементів:

$$(0, 1, 2)\ (0, 2, 3),\ (0, 3, 4) \tag{1.6}$$

Потік довжиною в n вершин дасть на виході n-2 трикутних примітивів даного типу. Незакінчений примітив буде проігнорований.

1.5.2.3 Об’єкти OpenGL ES 3.0

Об'єкт OpenGL (OpenGL Object) [29] — конструкція в поняттях OpenGL, що має певний стан. Коли конкретний об'єкт приєднаний до контексту OpenGL, його станом можна маніпулювати за допомогою стану самого контексту. Таким чином, зміни, що вносяться до стану контексту OpenGL, будуть застосовані і до прив'язаного об'єкта, а функції, викликані на поточний стан контексту, використовуватимуть стан прив'язаного об'єкта.

OpenGL визначається як машина станів. Різні виклики методів API призводять до зміни стану машини OpenGL, його вилучення або вилучення його частин або використання поточного стану для рендерингу.

Об'єкти в контексті OpenGL завжди грають роль контейнерів стану. Кожен окремий тип об'єкта визначається конкретним станом, який останній містить. Об'єкт в OpenGL — спосіб інкапсуляції певної групи станів і можливості їх зміни за допомогою виклику однієї функції.

Слід розуміти, що наведене вище визначено специфікацією OpenGL. Сама реалізація на рівні драйвера залишається невизначеною, але при цьому, найчастіше, не впливає на поведінку і взаємодію з станами, як це наведено в специфікації.

1.5.2.3.1 Створення і видалення об’єктів OpenGL

У процесі створення об'єкта ключову роль відіграє генерація його цілочисельного імені. Це — свого роду посилання на об'єкт. Проте, ця процедура необов’язково задає стан створеного об'єкта. Для більшості типів OpenGL об'єктів, новий об'єкт буде містити стан, визначений за замовчуванням. Функції для генерації імені об'єкта мають загальний вигляд наступного типу:

$$glGen*, \tag{1.7}$$

де * — тип об'єкту. Всі функції такого типу мають однакову сигнатуру:

$$void\ glGen*(GLsizei\ n,\ GLuint\ *objects); \tag{1.8}$$

Така функція генерує n об'єктів даного типу, зберігаючи їх імена в масив, наданий параметром *objects*. Це дозволяє створювати безліч об'єктів за допомогою одного виклику.

Ім'я об'єкта завжди представляє цілочисельне значення. Не слід сприймати ці імена як справжні покажчики, вони є номерами, які ідентифікують об'єкт. Ідентифікатор зі значенням 0 зарезервований. Всі інші невід'ємні 32-бітові значення доступні для іменування.

Після використання об'єкта необхідно видалити його. Для цього існує набір функцій, що має загальний вигляд наступного типу:

void glDelete(GLsizei n, const* GLuint **objects);* (1.9)

Дані функції працюють подібно glGen-функціям, за винятком того, що видаляють об'єкт замість його створення. Не валідні значення покажчиків будуть проігноровані цією функцією.

1.5.2.3.2 Використання об’єктів OpenGL

З огляду на визначення OpenGL як машини станів, для модифікації існуючих об'єктів необхідно спочатку прив'язати їх до поточного контексту. Прив'язка об'єкта до контексту встановлює стан контекста об'єкту. Таким чином, будь-яка функція, покликана змінювати стан об'єкта буде застосована до стану прив'язаного до контексту об'єкта.

Прив'язка нового об'єкта як правило створює цьому об'єкту новий стан. Різні типи об'єктів мають різні функції для прив'язки. Всі вони мають загальний вигляд наступного типу:

void glBind(GLenum target, GLuint object);* (1.10)

де * — тип об'єкта, а *object* — ім'я об'єкта.

target — ціль прив'язки. Деякі типи об'єктів можуть бути прив'язані до безлічі цілей, в той час, як інші — тільки до однієї певної. Наприклад, об'єкт-буфер може бути прив'язаний як масив-буфер, буфер індексів, буфер пікселів і т.д. Різні цілі

приймають різні прив'язки. Таким чином, можливо прив'язати один об'єкт-буфер як масив вершинних атрибутів, а інший — як буфер індексів.

Якщо об'єкт прив'язується до цілі, до якої вже прив'язаний інший об'єкт, то попередній прив'язаний об'єкт відв'язується від даної цілі.

1.5.2.3.3 Об’єкт-буфер (Buffer Object)

Buffer Object — об'єкт в поняттях OpenGL, який зберігає масив невпорядкованою пам'яті, виділеної контекстом OpenGL. Він може використовуватися для зберігання вершинних даних, значень пікселів з зображень чи кадрового буфера і т.д.

Buffer Object є OpenGL об'єктом, таким чином до нього застосовні всі ті ж правила, що і для першого. Для створення Buffer Object використовується функція glGenBuffers. Для видалення — glDeleteBuffers. Вони — частина стандартної парадигми роботи з більшістю OpenGL об'єктів.

Дотримуючись стандартної парадигми OpenGL, використовується наступна функція прив'язки Buffer Object:

void glBindBuffer(enum target, uint bufferName) (1.11)

де *target* визначає те, як буде використовуватися прив'язка об'єкта. Для створення і / або заповнення об'єкта-буфера даними мета не несе ніякого значення.

Об'єкт-буфер представляє лінійний масив в пам'яті довільного розміру. Пам'ять повинна бути виділена до використання буфера або завантаження даних в нього. У контексті даного дослідження розглянемо один із способів завантаження даних в об'єкт-буфер. Для даної мети використовується наступна функція API:

*void glBufferData(enum target, sizeiptr size, const void *data, enum usage)*(1.12)

де параметр *target* являє собою те ж саме, що і в контексті функції glBindBuffer. *size* представляє кількість байт для виділення поточному об'єкту-буферу. Параметр *data* є покажчиком на виділену користувачем пам'ять, звідки дані будуть скопійовані в об'єкт-буфер. Параметр *usage* більш комплексний, він буде розглянутий окремо.

Об'єкти-буфери — сховища пам'яті загального призначення, виділеної OpenGL. Існує безліч варіантів їх використання. Для забезпечення гнучкості рішення і високої продуктивності використовується параметр *usage*. Він являє узагальнений опис того, як користувач збирається використовувати конкретний об'єкт-буфер.

Існують дві незалежні частини шаблону використання даного параметра: те, як користувач збирається читати і писати з / в буфер і те, як часто користувач збирається модифікувати буфер у відношенні до частоти використання останнього.

Існують два способи модифікації вмісту буфера: явне завантаження нових даних користувачем і непряма модифікація за допомогою залучених OpenGL команд.

Аналогічно, користувач має можливість зчитувати дані буфера використовуючи широкий набір команд. Або ж користувач має можливість викликати команду, яка побічно призведе до читання буфера за допомогою OpenGL. Наприклад, буфери, які зберігають дані вершин читаються конвеєром OpenGL при рендерингу.

Існують три підказки-параметра, які визначають те, що користувач збирається робити з буфером:

- DRAW — користувач буде записувати дані в буфер, але не буде зчитувати їх.
- READ — користувач буде зчитувати отримані дані, але не буде записувати їх в буфер.
- COPY — користувач не буде ні зчитувати, ні записувати дані.

Очевидно, що DRAW-підказка корисна для буферів, за допомогою яких буде здійснюватися рендеринг. В даному випадку користувач тільки завантажує дані в буфер, а зчитуванням займається OpenGL.

READ-підказка використовується у випадках, коли буфер використовується як сховище результатів виконання OpenGL команд.

COPY-підказка використовується у випадку, якщо буфер буде використовуватися для передачі даних в рамках конвеєра OpenGL і клієнтського додатку. Наприклад, користувач може зчитати дані зображення в такий буфер, а потім — використовувати його для рендеринга. В даному випадку користувач взагалі не здійснює явного читання і / або запису в буфер.

Існують три підказки-параметра для опису частоти використання даних в буфері:

- STATIC — припускає одноразову установку вмісту буфера.
- DYNAMIC — припускає можливість випадкової, багаторазової установки.
- STREAM — припускає модифікацію даних буфера після кожного або майже кожного використання.

Підказка STREAM легка для розуміння: зміст буфера буде оновлюватися практично після кожного використання. STATIC-підказка також не створює труднощів: вміст буфера буде завантажений один раз і ніколи не буде модифікований.

Труднощі виникають в ситуаціях, коли буфер з DYNAMIC-підказкою стає STREAM- або STATIC-буфером. Також слід зазначити, що OpenGL все одно може модифікувати STATIC-буфер або ніколи не модифікувати STREAM-буфер. Аналогічно, до труднощів відносяться питання ефективності використання STATIC-буферів для вмісту, який буде оновлюватися рідко, ефективності використання DYNAMIC-буферів для відносно частого оновлення, але не в порядках частоти оновлення STREAM-буфера, ефективності використання DYNAMIC-буферів для частково оновлюваного вмісту. Всі ці

нюанси можливо уточнити, вдаючись до ретельного профілювання готових рішень на цільових платформах.

Також слід розуміти, що STREAM-, STATIC-, і DYNAMIC-підказки поєднуються в будь-яких комбінаціях з READ-, DRAW-, і COPY-підказками.

Як було зазначено вище, можливо використовувати функцію glBufferData для поновлення даних в буфері. Реалізація даної функції передбачає повторне виділення пам'яті для буфера. Таким чином, вона погано підходить для простого оновлення змісту буфера. Замість неї, для часткового поновлення буфера можливо використовувати наступну функцію:

*void glBufferSubData(enum target, intptr offset, sizeiptr size, const void *data)*(1.13)

Тут, параметр *offset* — ціле число, яке представляє зміщення в буфері, з позиції якого необхідно виконувати оновлення. Параметр *size* відповідає за кількість байт, які будуть скопійовані з *data*.

1.5.2.4 Об’єкти GLSL

GLSL Об'єкт [30] — об'єкт в поняттях OpenGL, який інкапсулює скомпільовані шейдери. Ці об'єкти представляють програмний код, написаний на мові OpenGL Shading Language (далі — GLSL). Дані об'єкти часто не відповідають парадигмі OpenGL об'єктів.

Об'єкт-програма (Program Object) представляє повністю оброблений виконавчий програмний код на певному етапі шейдера. Порожня програма-об'єкт створюється з використанням наступної функції:

GLuint glCreateProgram() (1.14)

Виклик цієї функції повертає порожню програму-об'єкт. Видалення програм відбувається з використанням функції glDeleteProgram. Порожні

об'єкти-програми повинні бути заповнені виконуваним кодом. Це відбувається за допомогою компіляції і компоновки шейдерів в програму.

Для використання програми необхідно прив'язати її до поточного контексту OpenGL. На відміну від OpenGL об'єктів, застосовується наступна функція:

void glUseProgram(GLuint program); (1.15)

Виклик цієї функції прив'яже *program* до поточного контексту, відв'язавши при наявності попередню об'єкт-програму.

1.5.2.5 Шейдери

Шейдер [31] представляє собою визначену користувачем програму, спроектовану для виконання на різних етапах графічного конвеєра. Завдання шейдера — виконати завдання програмованого етапу конвеєра.

Конвеєр визначає набір програмованих етапів. Кожен такий етап становить певний тип програмної обробки даних в конвеєрі.

Шейдера розробляються на мові GLSL. Далі будуть розглянуті застосовні в рамках системи типи шейдерів.

При рендерингу буде використана прив'язана до контексту програма-об'єкт. Її частини будуть виконані один або більше разів залежно від того, що рендерится. Кожен тип шейдера визначає частоту його виконання.

1.5.2.5.1 Вершинний шейдер

Вершинний шейдер [32] являє собою програмований етап конвеєра, який відповідає за обробку окремих вершин. На вхід вершинного шейдера надходять дані — вершинні атрибути.

1.5.2.5.2 Фрагментний шейдер

Фрагментний шейдер [33] являє собою програмований етап конвеєра, який відповідає за перетворення фрагментів, згенерованих на етапі растеризації у набір кольорів.

1.5.2.6 Компіляція шейдерів

Компіляція шейдера [34] — термін в контексті OpenGL, який використовується для опису процесу завантаження GLSL програм в OpenGL для використання в якості шейдерів. Об'єкт-програма може містити виконавчий код для всіх етапів шейдера, таким чином, все необхідне для рендеринга може бути прив'язане до однієї програми. Побудова програм, що містять кілька шейдерних етапів передбачає застосування двоетапного процесу компіляції.

Двоетапний процес компіляції відображає стандартний процес компіляції і компонування початкового коду C / C++. Початковий код подається на вхід компілятора, який на виході дає об'єктний модуль. Скомпонувавши кілька об'єктних модулів можливо отримати виконуваний код.

Перший крок компіляції шейдера — створення об'єктів-шейдерів за допомогою використання наступної функції:

GLuint glCreateShader(GLenum shaderType); (1.16)

В результаті виконання даної функції буде створено порожній об'єкт-шейдер для етапу шейдера, вказаного в секції *shaderType*. Даний параметр набуває таких значень:

- GL_VERTEX_SHADER,
- GL_GEOMETRY_SHADER,
- GL_FRAGMENT_SHADER.

Після створення об'єкта-шейдера необхідно надати валідний рядок, що містить початковий GLSL код за допомогою наступної функції:

void glShaderSource(GLuint shader, GLsizei count,
*const GLchar **string, const GLint *length);* (1.17)

Ця функція приймає масив рядків параметром *string* і зберігає його в певний *shader*. Будь-які попередньо збережені рядки будуть очищені. Параметр *count* представляє кількість окремих рядків. OpenGL скопіює надані рядки у внутрішню пам'ять.

Параметр *length* може приймати масив цілочисельних значень розмірністю в *count*. Він представляє довжини відповідних рядків в масиві *string*.

По установці рядків початкового коду в шейдер, стає можливою його компіляція за допомогою наступної функції:

void glCompileShader(GLuint shader); (1.18)

1.5.2.7 Конфігурація об’єкта-програми

Після успішної компіляції об'єктів-шейдерів стає можливою компоновка програми. Компонування починається з виклику наступної функції:

void glAttachShader(GLuint program, GLuint shader); (1.19)

Дану функцію можливо викликати безліч разів для різних об'єктів-шейдерів. Після компонування слід від'єднати всі об'єкти-шейдери від програми. Це проводиться наступною функцією:

void glDetachShader(GLuint program, GLuint shader); (1.20)

shader повинен бути прикріплений до зазначеної *program*.

Якщо відсутня потреба використовувати даний об'єкт-шейдер для компонування інших програм, можливо його видалити. Це здійснюється за допомогою функції glDeleteShader. Видалення шейдера буде відкладено до моменту від'єднання останнього від програми. Таким чином, слід від'єднувати шейдера після компонування.

1.5.2.7.1 Uniform-змінні (Uniform Variables)

Uniform [35] — глобальна GLSL змінна, оголошена з модифікатором «uniform». Представляє параметр, що користувач, який використовує програму-шейдер, може передати в останню. Зберігаються вони в програмі-об'єкті. Дані змінні незмінні від одного виконання шейдерної програми до іншого в рамках одного проходу рендерингу.

Uniform-змінні оголошуються як глобальні. Вони можуть мати будь-який підтримуваний або агрегований тип. Нижче наведено приклад блоку коректного GLSL початкового коду з використанням uniform-змінних (рисунок 1.14):

```
1  struct TheStruct
2  {
3    vec3 first;
4    vec4 second;
5    mat4x3 third;
6  };
7
8  uniform vec3 oneUniform;
9  uniform TheStruct aUniformOfArrayType;
10 uniform mat4 matrixArrayUniform[25];
11 uniform TheStruct uniformArrayOfStructs[10];
```

Рисунок 1.14 — Приклад блоку коректного початкового коду GLSL з використанням uniform-змінних

Uniform-змінні константні в рамках шейдера. Спроба їх модифікації призведе до помилки на етапі компіляції. Можлива ініціалізація даних змінних за допомогою стандартного GLSL синтаксису (рисунок 1.15):

```
1   uniform vec3 initialUniform = vec3(1.0, 0.0, 0.0);
```

Рисунок 1.15 — Приклад блоку коректного початкового коду GLSL з ініціалізацією uniform-змінних

Як і у випадку з OpenGL об'єктами, програмні об'єкти мають певний стан. В даному випадку, цей стан представляє набір uniform-змінних, які будуть використовуватися для візуалізації за допомогою даної програми. Всі етапи виконання програмного об'єкта використовують той же набір uniform-змінних. Таким чином, однойменна uniform-змінна в рамках вершинного і фрагментного шейдера представлятиме одне і те ж значення. Отже, оголошення uniform-змінних з однаковим ім'ям і різним типом в рамках двох різних шейдерів призведе до помилки компоновки програми. Кожна uniform-змінна має своє унікальне і довільне в рамках певної програми розташування. Основна мета визначення розташування uniform-змінних полягає в отриманні можливості модифікувати останні. Це здійснюється за допомогою glUniform * функцій, де * визначає тип завантажується в змінну значення. Такими можуть бути матриці, вектори, окремі значення і їх масиви. Для того, щоб модифікувати значення uniform-змінної, для початку необхідно прив’язати цільову програму до контексту за допомогою функції glUseProgram.

1.5.2.7.2 Змінні (Varying Variables)

Varying-змінні надають інтерфейс взаємодії шейдерів. Наприклад, вершинний шейдер розраховує значення для кожної окремої вершини, фрагментний — для кожного фрагменту. При визначенні varying-змінної в

вершинному шейдері, її значення буде інтерполюватися в рамках примітиву, який зараз рендериться таким чином, що буде можливо отримати інтерпольоване значення у фрагментному шейдері. Varying-змінні можливо використовувати з float, vec2, vec3, vec4, mat2, mat3, mat4 типами, а також їх масивами.

1.5.2.8 Z-буферизація

Depth Test [36] представляє собою пофрагментну операцію обробки в рамках фрагментного шейдера. Вихідні значення глибини фрагмента можливо порівняти з глибиною семплу, в який відбувається запис. Якщо порівняння повертає негативний результат — фрагмент відкидається. В іншому випадку — буфер глибини (Z-буфер) буде оновлено з урахуванням нової вихідної глибини за винятком ситуації, коли певна операція або конфігурація конвеєра запобіжить запису в даний буфер.

Для включення Depth Test необхідно викликати функцію glEnable з параметром-ключем GL_DEPTH_TEST. Під час рендерингу в кадровий буфер без Z-буфера тест буде видавати результати відповідні відключеному Depth Test.

Дана операція буде також використовуватись в даному дослідженні.

РОЗДІЛ 2. ПРОЕКТУВАННЯ РЕАЛІЗАЦІЇ СТЕКУ ТЕХНОЛОГІЙ

2.1 Вершинний буфер, VBO

Vertex Buffer Object — об'єкт-буфер в контексті OpenGL, який використовується в якості джерела вершинних даних. В особливості його функціонування немає принципових відмінностей від будь-якого іншого об'єкта-буфера.

Для використання Vertex Buffer Object як джерела вершинних даних в першу чергу необхідно прив'язати його до GL_ARRAY_BUFFER цілі. Далі

відбувається «розмітка» буфера за допомогою різних імплементацій-перевантажень функції glVertexAttribPointer:

void glVertexAttribPointer(GLuint index, GLint size, (2.1)
*GLenum type, GLboolean normalized, GLsizei stride, const void *offset);*

Ці функції вказують, що за допомогою індексу атрибуту, переданого параметром *index* можливо отримати відповідні йому атрибутні дані з будь-якого прив'язаного до GL_ARRAY_BUFFER об'єкта-буфера.

2.2 Формат вершин

Функції glVertexAttribPointer визначають положення даних атрибута по його індексу. Також вони визначають те, як OpenGL повинен інтерпретувати дані. Таким чином, ці функції виконують дві концептуальні ролі: встановлюють буферу інформацію про джерела даних і визначають формат останніх.

Параметри форматування описують те, як необхідно інтерпретувати дані про одну вершину з масиву. Вершинні атрибути у вершинному шейдері можуть бути оголошені як GLSL типи з плаваючою комою (такі, як float або vec4), цілочисельні типи (такі, як uint або ivec3), типи подвійної точності (такі, як double або dvec4).

Тип атрибута в вершинному шейдері повинен відповідати типу, що надається масивом атрибутів. Кожен атрибутний індекс представляє вектор певного типу довжиною від 1 до 4 компонент. Параметр *size* функції glVertexAttribPointer визначає кількість компонент у векторі, що надається масивом атрибутів. Слід зазначити, що в разі, коли вершинний шейдер вимагає меншої кількості компонент, ніж надає масив атрибутів, зайві компоненти будуть проігноровані. У зворотному випадку — компоненти доповнюються з вектора виду (0, 0, 0, 1) для відсутніх XYZW компонент.

2.2.1 Типи компонентів

Тип компонента [37] вектора в об'єкті-буфері надається параметрами *type* і *normalized*. Різні варіанти функції glVertexAttribPointer приймають різні типи в якості параметру. Далі будуть наведені основні використовувані типи для функції glVertexAttribPointer:

- Типи з плаваючою комою. Параметр *normalized* повинен бути GL_FALSE
  - GL_FLOAT — 32-бітове значення одинарної точності з плаваючою комою.
  - GL_DOUBLE — 64-бітове значення подвійної точності з плаваючою комою.
  - GL_FIXED — 16-бітове значення з фіксованою комою.
- Цілочисельні типи. Перетворюються в типи з плаваючою комою автоматично. Якщо параметр *normalized* приймає значення GL_TRUE, то перетворення буде виконано за допомогою нормалізації числа. Якщо ж *normalized* приймає значення GL_FALSE, то значення буде перетворено безпосередньо, як це відбувається в Сі-подібному приведення типів.
  - GL_BYTE — знакове 8-бітове значення в додатковому коді.
  - GL_UNSIGNED_BYTE — беззнакове 8-бітове значення.
  - GL_SHORT — знакове 16-бітове значення в додатковому коді.
  - GL_UNSIGNED_SHORT — беззнакове 16-бітове значення.
  - GL_INT — знакове 32-бітове значення в додатковому коді.
  - GL_UNSIGNED_INT — беззнакове 32-бітове значення.

2.3 Офсет і шаг по індексу

Інформація, описана вище, використовується для того, щоб вказати OpenGL, як слід інтерпретувати дані. Формат вказує на розмірність кожної вершини в байтах і спосіб їх перетворення в значення атрибуту, яке приймає

шейдер. OpenGL вимагає ще два важливі елементи інформації для навігації по даним. Перший елемент — офсет з початку об'єкта-буфера до першого елемента його масиву. Другий — крок за індексом, який представляє кількість байт з початку одного елемента в масиві до початку іншого. Крок за індексом використовується для визначення байтової відстані між вершинами. Якщо параметр *stride* встановлений в 0, OpenGL буде сприймати масив даних як «тісно» упакований. Таким чином, встановивши *size* в 3, а *type* — в GL_FLOAT, OpenGL розрахує крок в 12 байт: 4 на значення з плаваючою комою і 3 таких значення на атрибут.

2.4 Атрибути, що перемежовуються

Основна мета кроку за індексом полягає в забезпеченні можливості перемежовування між різними атрибутами. Цю особливість легко продемонструвати таким блоком початкового коду C (рисунок 2.1):

```
10  struct Vertex
11  {
12    GLfloat position[3];
13    GLfloat normal[3];
14    Glubyte color[4];
15  };
16
17  Vertex vertices[VERTEX_COUNT];
```

Рисунок 2.1 — Блок початкового коду C, що демонструє перемежування вершинних атрибутів

В даному прикладі структура *structOfArrays* представляє кілька масивів елементів. Кожен масив «тісно» упакований, кожен масив незалежний від іншого. Змінна *vertices* являє собою єдиний масив, в якому кожен елемент представляє незалежну вершину.

Якщо створити об'єкт-буфер, який приймає масив *vertices*, а *baseOffset* прийняти за початок даних в даному масиві, то крок за індексом *stride* дозволить OpenGL отримати доступ до всіх потрібних атрибутів наступним способом (рисунок 2.2):

```
glVertexAttribPointer(
    0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
    reinterpret_cast<void*>
    (baseOffset + offsetof(Vertex, position))
);
glVertexAttribPointer(
    1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
    reinterpret_cast<void*>
    (baseOffset + offsetof(Vertex, normal))
);
glVertexAttribPointer(
    2, 4, GL_UNSIGNED_BYTE, GL_TRUE, sizeof(Vertex),
    reinterpret_cast<void*>
    (baseOffset + offsetof(Vertex, color))
);
```

Рисунок 2.2 — Блок початкового C++ коду, що демонструє використання перемежованих вершинних атрибутів

Слід зазначити, що всі атрибути використовують однаковий крок за індексом — розмірність Vertex структури. Таким чином, розмірність даної структури точно представляє кількість байт з початку одного елемента до початку іншого для кожного атрибута. Макрос *offsetof* розраховує байтовий офсет даного поля структури. Отриманий офсет підсумовується з *baseOffset*, таким чином кожне поле вказує на початок власних даних відносно початку Vertex структури. Загальне правило хорошого тону OpenGL — повсюдне використання описаного підходу.

Також слід зазначити, що всі наведені в цілях загального ознайомлення приклади виконані на мові програмування C / C++, докладний розгляд принципів і технік якого не входить в рамки дослідження даної роботи.

На наступних схемах (рисунок 2.3) будуть зображені базові підходи щодо використання вершинного буфера в частині заповнення останнього вершинними даними і їх упорядкування. Виходячи з описаного вище, в стеці технік, що досліджується та розробляється буде використовуватися третій із зображених підходів.

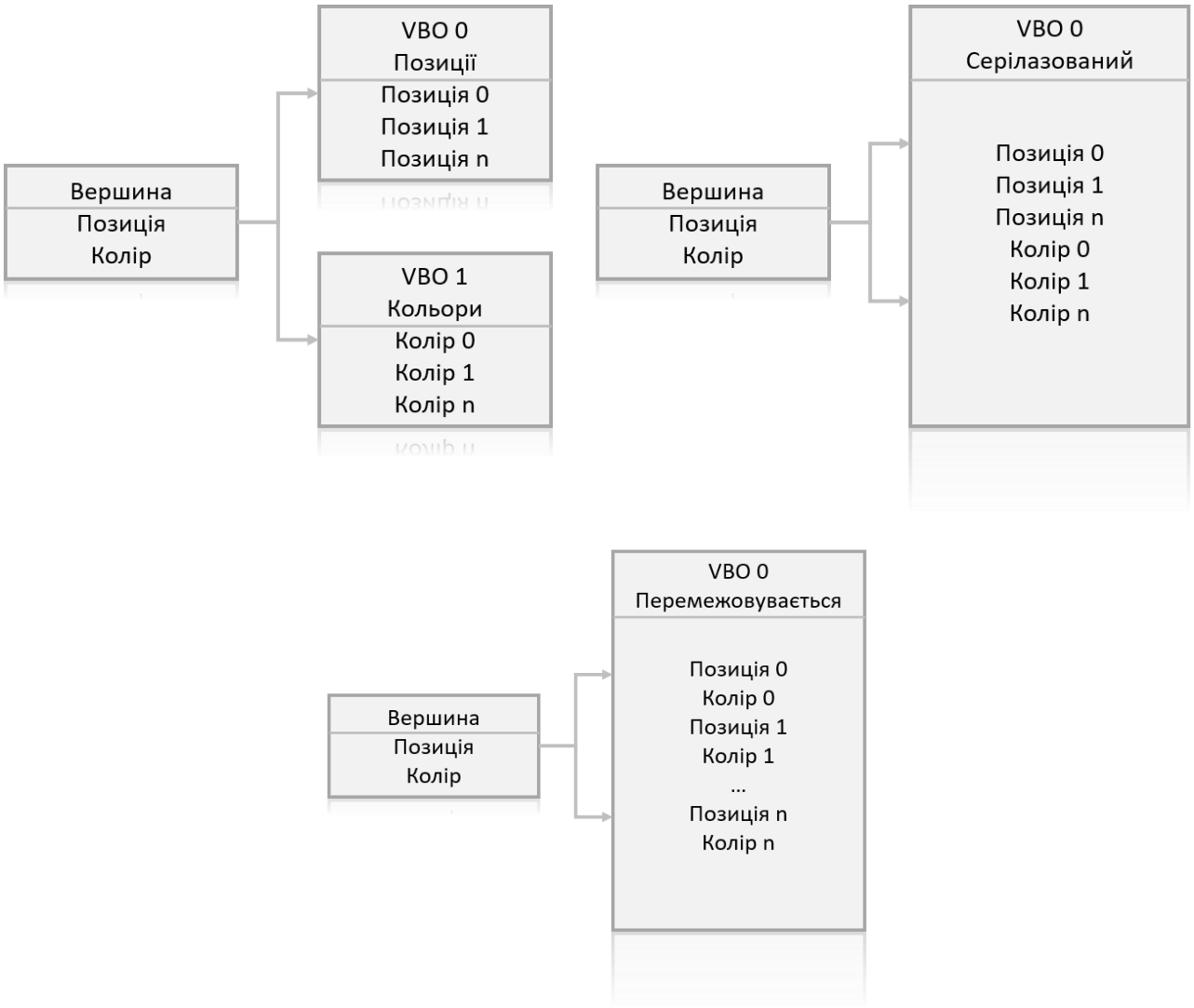


Рисунок 2.3 — Схеми базових підходів до використання вершинного буфера в частині заповнення останнього вершинними даними і їх упорядкування

2.5 Індексний буфер, IBO

Індексований рендеринг, як це було помічено в підрозділі 1.5.2.1, вимагає наявності масиву індексів. Всі вершинні атрибути будуть використовувати один і той же індекс з даного масиву. Даний масив може бути надано за допомогою об'єкта-буфера, прив'язаного до GL_ELEMENT_ARRAY_BUFFER. При такій прив'язці все команди рендеринга виду gl * Draw * Elements * використовуватимуть індекси з даного буфера. В якості індексів можливо використовувати беззнакові типи даних: byte, short, int. На наступній схемі (рисунок 2.4) ілюстративно зображений порядок і ефект використання розглянутих функцій.

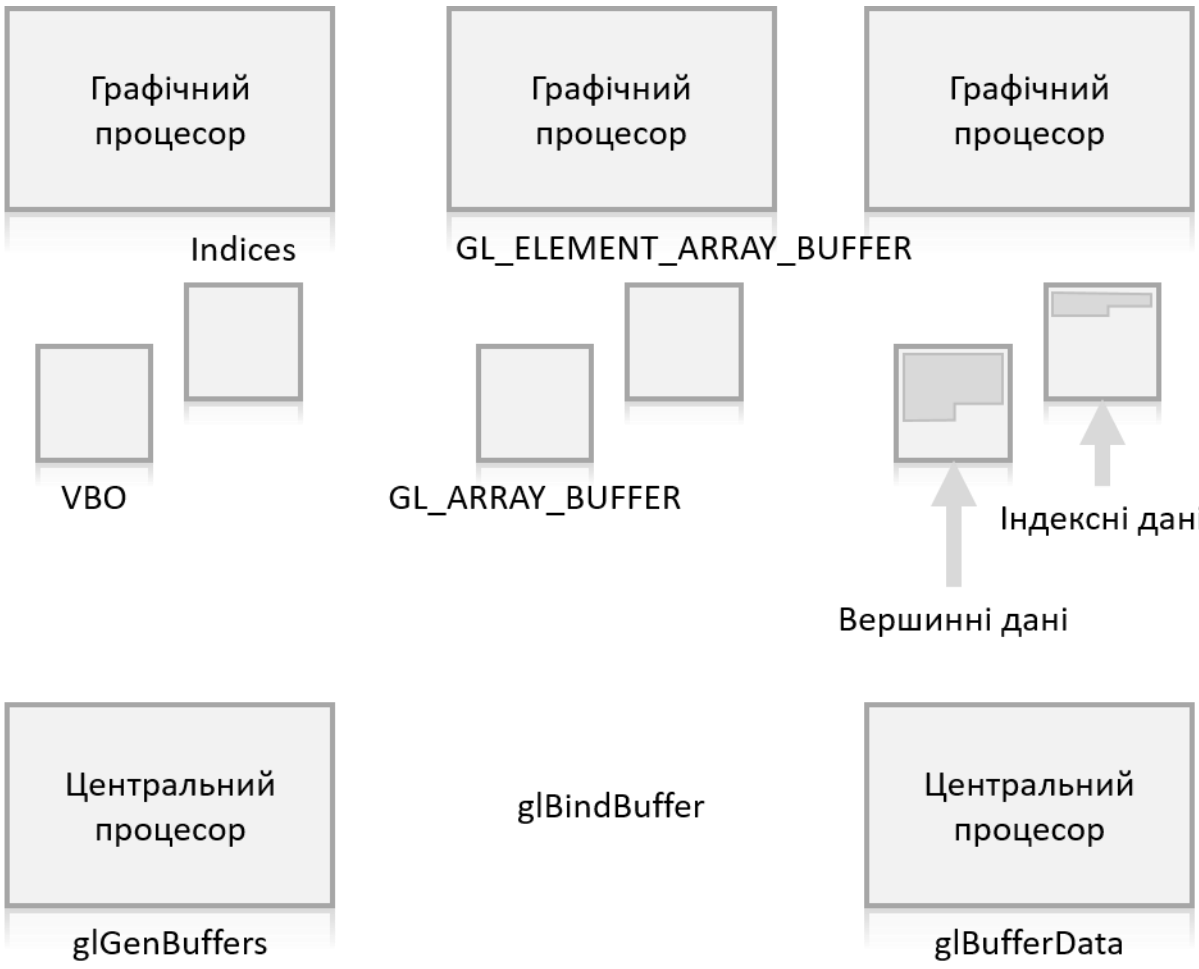


Рисунок 2.4 — Схема послідовності використання функцій

2.6 Пакетування примітивів

Описаний далі підхід сам по собі дозволяє збільшити продуктивність рендеринга і надає можливості проектування і розробки ряду інших підходів, заснованих на даному.

Як вже було багато разів зазначено в попередніх розділах, одним з вузьких місць ефективного тривимірного рендерингу в досліджуваному середовищі є проблема низької продуктивності передачі даних додатка в конвеєр OpenGL. Ця проблема здебільшого викликана великою кількістю змін стану конвеєра і існуванням великих накладних витрат виклику нативних функцій інтерфейсу OpenGL з середовища додатку в класичному підході до організації даної передачі.

Один із напрямків дослідження — можливості мінімізації кількості змін стану конвеєра і кількості викликів нативних функцій інтерфейсу за допомогою об'єднання геометрії в групи. Таким чином, передбачається створення буфера вершинних атрибутів для безлічі примітивів на стороні додатка та їх пакетна відправка в OpenGL конвеєр за допомогою відповідної попередньої конфігурацій останнього.

2.7 Добуток матриць афінних перетворень

Афінне перетворення представляє відображення [38]

$$f : \mathbf{R}^n \to \mathbf{R}^n, \tag{2.1}$$

що можна записати у вигляді

$$f(x) = M \cdot x + \upsilon, \tag{2.2}$$

де M — невироджена матриця і

$$\upsilon \in \mathbf{R}^n \tag{2.3}$$

Афінні перетворення зазвичай використовуються у лінійній алгебрі для представлення лінійних перетворень, а векторна сума — для представлення паралельних перенесень. За допомогою розширеної матриці можливо представити і те, і те як матричний добуток. Ця техніка вимагає розширити всі вектори додаванням «1» в кінці, всі матриці розширюються додаванням рядка нулів знизу, і колонки — вектору переносу — справа, а також одиниці в нижній правий кут. Якщо A матриця,

$$\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \begin{bmatrix} A & \vec{b} \\ 0,\ldots,0 & 1 \end{bmatrix} \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix} \tag{2.4}$$

те саме, шо

$$\vec{y} = A\vec{x} + \vec{b} \tag{2.5}$$

Зазвичай матрично-векторний добуток завжди відображає початок координат на початок координат, і, таким чином, не може представляти перенесення, яке обов'язково переносить початок координат в іншу точку. Додаванням «1» до кожного вектору, вважаємо простір відображеним на підмножину простору з одним додатковим виміром. В цьому просторі, початковий простір займає підмножину в якій останній індекс 1. Таким чином початок координат початкового простору буде знаходитися в *(0,0, ... 0, 1)*. Перенесення всередині початкового простору в термінах лінійного перетворення простору з більшою кількістю вимірів стає можливим. Це є приклад однорідних координат [39, 40].

Перевагою використання однорідних координат є те, що можливо комбінувати будь-яку кількість перетворень в одне шляхом перемноження матриць. Ця можливість використовується графічними програмами.

Матриця виду описує представлення афінного перетворення як-то стиснення, розтягнення, поворот, паралельний перенос, відображення та інші окремі та загальні випадки трансформацій.

Простір виду представляє результат перетворення світових координат в координати видимого простору. Видимий простір в даному випадку визначається сукупністю перетворень здвигів і поворотів сцени. Ці комбіновані перетворення представляються матрицею виду.

Сучасні системи рендерингу двовимірної графіки також оперують поняттям простору відсічення. Простір відсічення визначає видимі області сцени і, як наслідок, вершини [41].

Загальний процес використання перетворень зображений на рисунку 2.5:

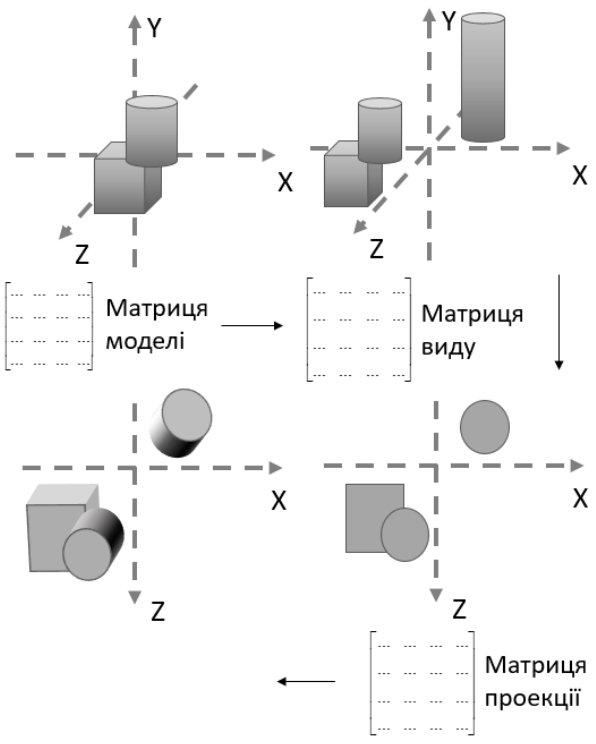


Рисунок 2.5 — Загальний процес використання матриць перетворень

2.8 Поточне оновлення буферів

Buffer Object Streaming [42] — процес частого поновлення об'єктів-буферів новими даними по мірі використання даних буферів. Потокове поновлення відбувається в наступному порядку: зі внесенням модифікацій в об'єкт-буфер, команда OpenGL проводить зчитування з нього, далі процес повторюється. OpenGL гарантує працездатність потокового оновлення, але не забезпечує високої продуктивності поновлення «з коробки».

Специфікація OpenGL дозволяє реалізації відкладати виконання команд рендерингу. Це дозволяє викликати рендеринг безлічі об'єктів, а потім дозволяти конвеєру OpenGL самому визначати час початку рендерингу останніх. З цієї причини цілком можливе виникнення ситуації, коли клієнтський код намагається оновити об'єкт-буфер, вже запропонований конвеєру OpenGL, але ще не використаний ним. Це призведе до задіяння механізмів синхронізації, які змусять викликаючий протік очікувати негайного виконання конвеєром команд, в яких задіяно використання даного об'єкта-буфера. Існує ряд стратегій для вирішення даної проблеми. Кожна зі стратегій має свої переваги і недоліки.

2.8.1 Явна множинна буферизація (Explicit multiple buffering)

Дане рішення досить просте. Воно передбачає створення двох паралельних лінійок об'єктів-буферів однакової довжини. У міру використання конвеєром одного буфера — можлива модифікація другого, потім — навпаки. Залежно від паралелізму, що надається конкретною імплементацією можливо збільшити кількість рядів буферів-об'єктів. Проблема даного рішення полягає в необхідності управління декількома рядами об'єктів-буферів, що, однак, може пошкодити технікам кешування.

2.8.2 Перевизначення буфера (Buffer re-specification)

Дане рішення полягає в повторному виділенні об'єкта-буфера перед його модифікацією. Цей підхід також називається Buffer Orphaning. В рамках даного дослідження існує відповідна функція для цієї мети — glBufferData з NULL параметром даних і точно тим же розміром і підказками використання, що і раніше. Це дозволяє реалізації просто повторно виділити простір для сховища поточного буфера за допомогою внутрішніх механізмів. Передбачається, що повторне виділення сховища об'єкта-буфера відбувається швидше, ніж його неявна синхронізація. Старе сховище буде і далі використано для посланих раніше команд, пов'язаних з читанням даного об'єкта-буфера. Передбачається, що при багаторазовому повторному виконанні описаної вище процедури, драйвер OpenGL і зовсім перестане виділяти пам'ять під нові дані, а буде лише поміщати використані блоки в чергу і повторно заповнювати і зчитувати в / з них. Очевидно, конкретна імплементація не визначена і ця поведінка нічим не гарантована, але передбачається її висока продуктивність. Дане рішення також буде застосовано і детально профільоване.

2.9 Нативні операції

Один із аспектів дослідження полягає в використанні нативних C / C ++ (JNI) функцій на стороні додатку в частині операцій з буферами даних і матриць афінних перетворень для графічного конвеєра на стороні додатку. Нативні функції імовірно забезпечують більшу продуктивність виконання, на відміну від реалізованих за допомогою Android / Java, незважаючи на накладні витрати ресурсів для їх виклику. Деталі використання нативних операцій в рамках даного дослідження буде наведено нижче. [43]

2.10 Очікувані результати

Виходячи з обраного стеку тривимірного рендерингу, технік та особливостей, очікується отримати позитивні показники продуктивності в результаті їх застосування, імплементації та конфігурації в частині ресурсів оперативної пам’яті, центрального та графічного процесорів у порівнянні з іншими розглянутими підходами та існуючими рішеннями.

РОЗДІЛ 3. ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ ТА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПРОПОНОВАНОГО РІШЕННЯ

3.1 Підготовка тестових моделей

В якості основної тривимірної моделі для дослідження був обраний стенфордський дракон (Stanford dragon) [44].

Деякі дані про модель:

- Джерело: Stanford University Computer Graphics Laboratory
- Сканер: Cyberware 3030 MS + spacetime analysis
- Загальний розмір сканувань: 2748318 точок (приблизно 5500000 трикутників)
- Розміри відтворення: 566098 вершин, 1132830 трикутників

На наступному рисунку зображений загальний вигляд моделі в середовищі відкритого програмного пакету для створення тривимірної комп'ютерної графіки, що включає засоби моделювання, анімації, вимальовування, після-обробки відео, а також створення відеоігор — Blender [45] (рисунок 3.1):

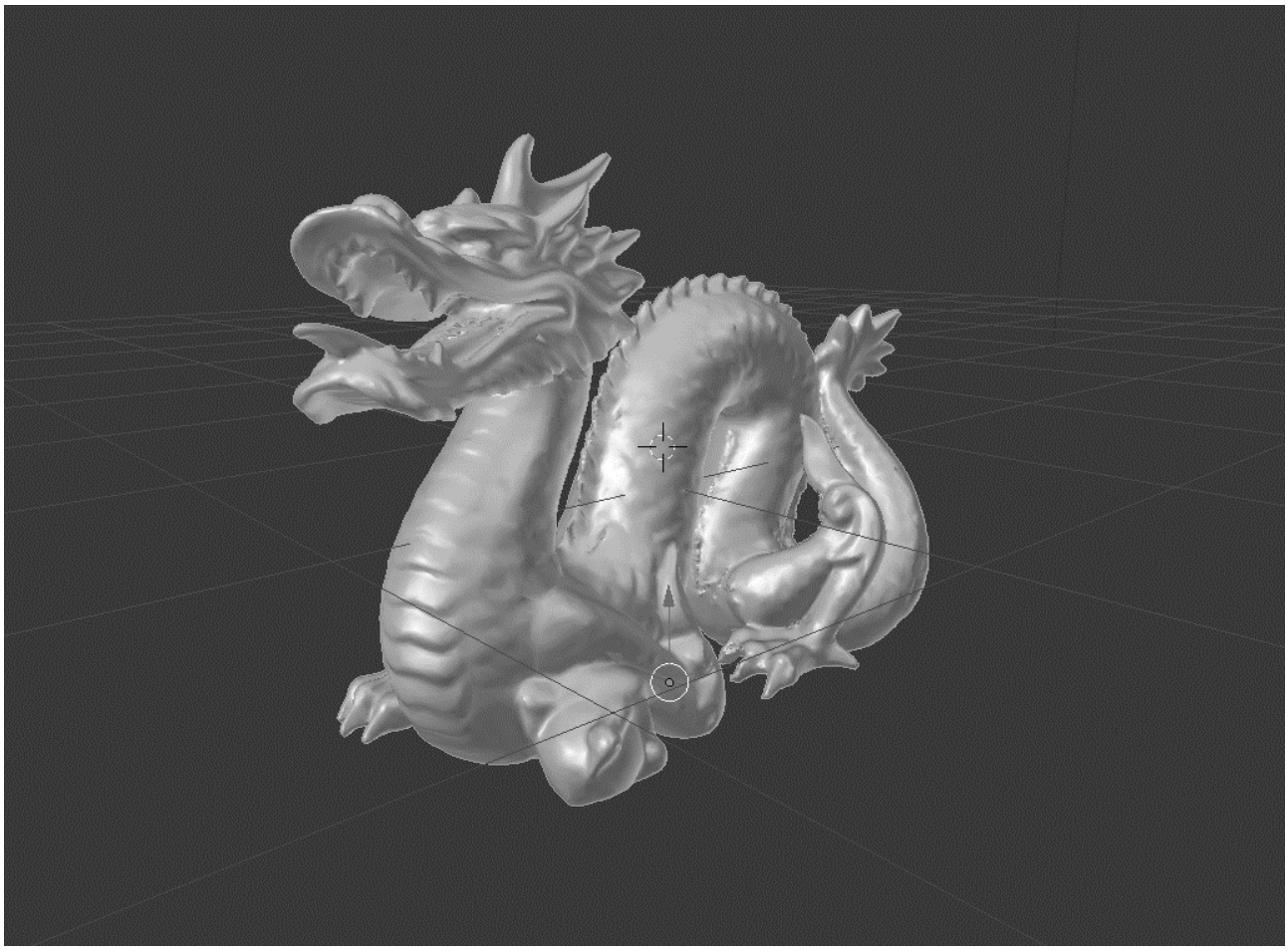


Рисунок 3.1 — Робоча зона Blender з тестовою моделлю

З плином імплементації та профілювання обраних підходів також були використані й інші тривимірні моделі, як-то: тор, ікосфера, UV-сфера, а також варіації стенфордського дракона з 100000 примітивів. Слід відмітити, що оригінальна модель дракона містить 871414 примітивів, а не 1132830, як зазначено в технічних характеристиках. На наступному рисунку зображений детальний вид поверхні моделі в режимі каркасу (рисунок 3.2):

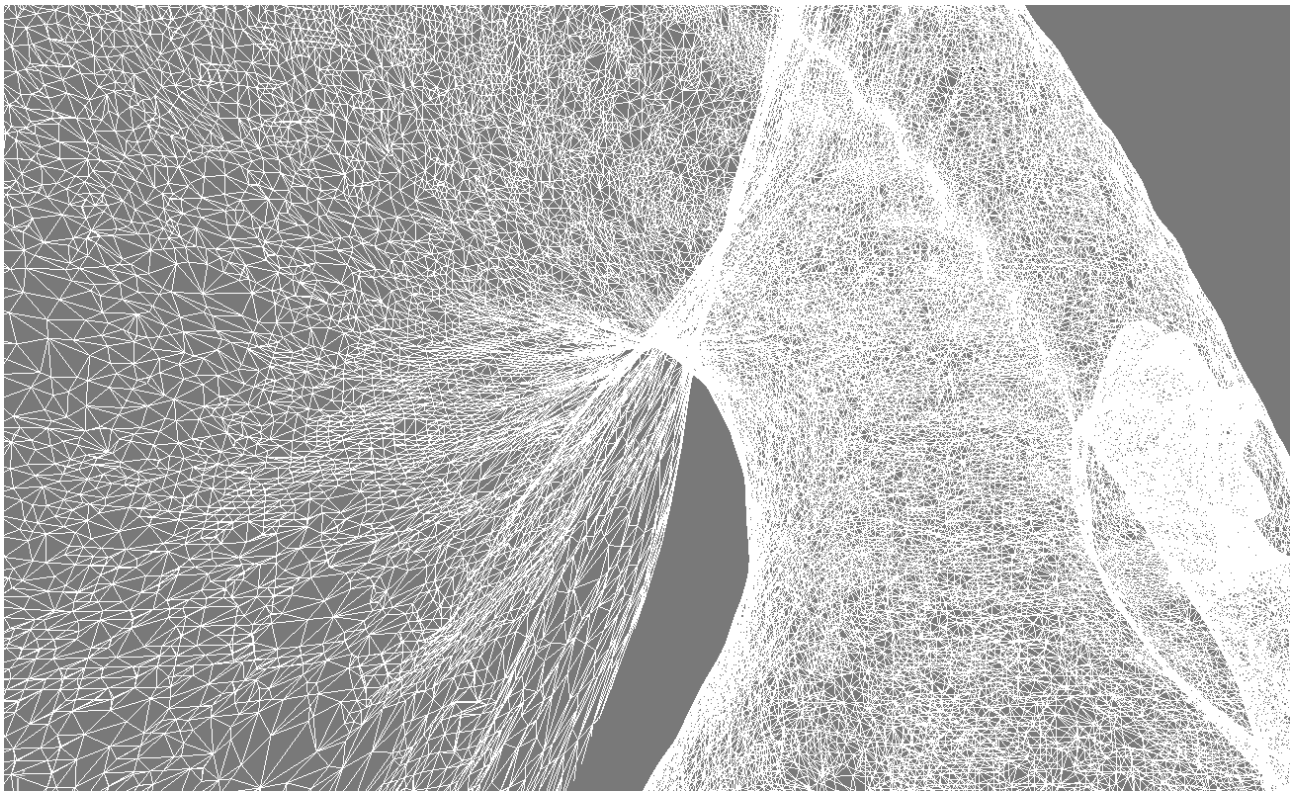


Рисунок 3.2 — Детальний вид поверхні моделі в режимі каркасу

3.2 Зневадження викликів OpenGL ES API, GAPID

GAPID — це набір інструментів, який дозволяє перевіряти, налаштовувати та відтворювати виклики додатка до графічного драйвера. [46]

На рисунку 3.3 проілюстрована частина інтерфейсу зневадження GAPID, що відображає екранні буфери окремих, цілих кадрів.

На рисунку 3.4 та 3.5 зображені окремо вибрані кроки рендерингу моделі на 100000 примітивів, проваджені GAPID. Можна наочно побачити результати рендерингу кожного наступного буферу вершин різної розмірності.

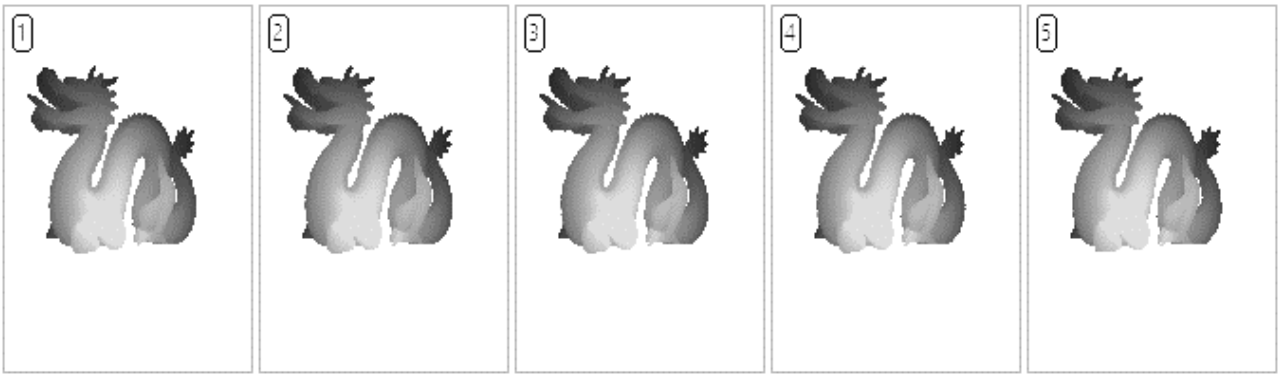


Рисунок 3.3 — Екранні буфери окремих кадрів

Рисунок 3.4 — Окремі етапи рендерингу буферів вершин

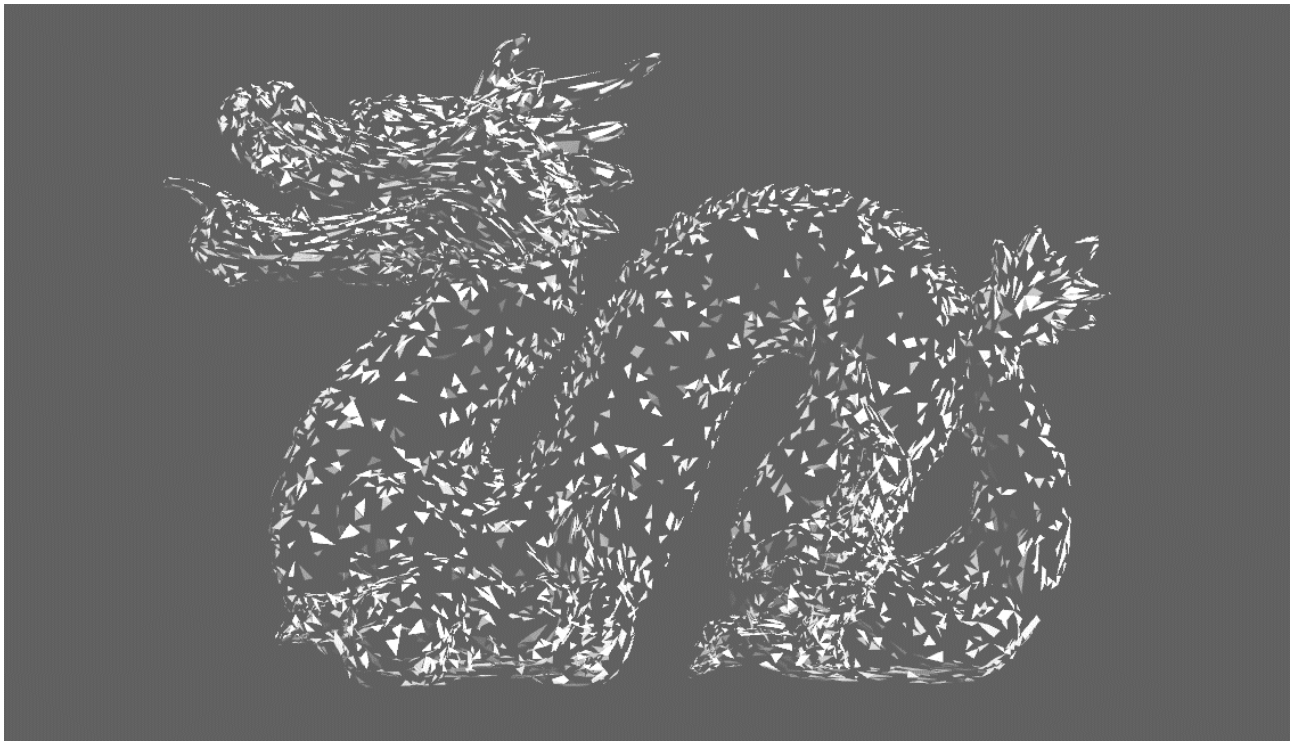


Рисунок 3.5 — Окремий етап рендерингу буферу вершин

3.3 Профілювання реалізації, Qualcomm® Snapdragon™ Profiler

Snapdragon Profiler — програмне забезпечення профілювання, яке працює на платформах Windows, Mac та Linux. Воно з'єднується з пристроями Android, оснащеними процесорами Snapdragon по USB. Snapdragon Profiler дозволяє розробникам аналізувати центральний, графічний процесор, DSP, пам'ять, потужність, теплові та мережеві дані, щоб розробники могли знаходити та виправляти вузькі місця роботи додатків. [47]

Дозволяє перегляди метрики в режимі реального часу, легко корелювати використання ресурсів системи на часовій шкалі, проваджує більш ніж 150 різних метрик продуктивності апаратного забезпечення у 22 категоріях.

Режим «Trace Capture» дозволяє візуалізувати ядра та системні події на часовій шкалі для аналізу системних подій на низькому рівні у контексті центрального, графічного процесорів та DSP.

Режим «Snapshot Capture» дозволяє зафіксувати та налагодити рендеринг кадрів будь-якого додатка OpenGL ES, відтворювати команди API, переглядати та редагувати шейдери.

Підтримує: OpenGL ES 3.1, OpenCL 2.1 та Vulkan 1.0 під процесором Snapdragon 820 (або пізніший), Android N (або пристрій Android 6.0 з графічним драйвером, який підтримує Vulkan).

Snapdragon Profiler — один із головних інструментів профілювання в рамках даного дослідження. За його допомогою були зняті та, в подальшому, оброблені і проаналізовані десятки метрик, що, в кінцевому результаті дали змогу робити висновки про успішність та ефективність виконаних в рамках дослідження розробок.

На рисунках 3.6 та 3.7 зображений загальний вигляд інтерфейсу Snapdragon Profiler в ході профілювання в режимі реального часу.

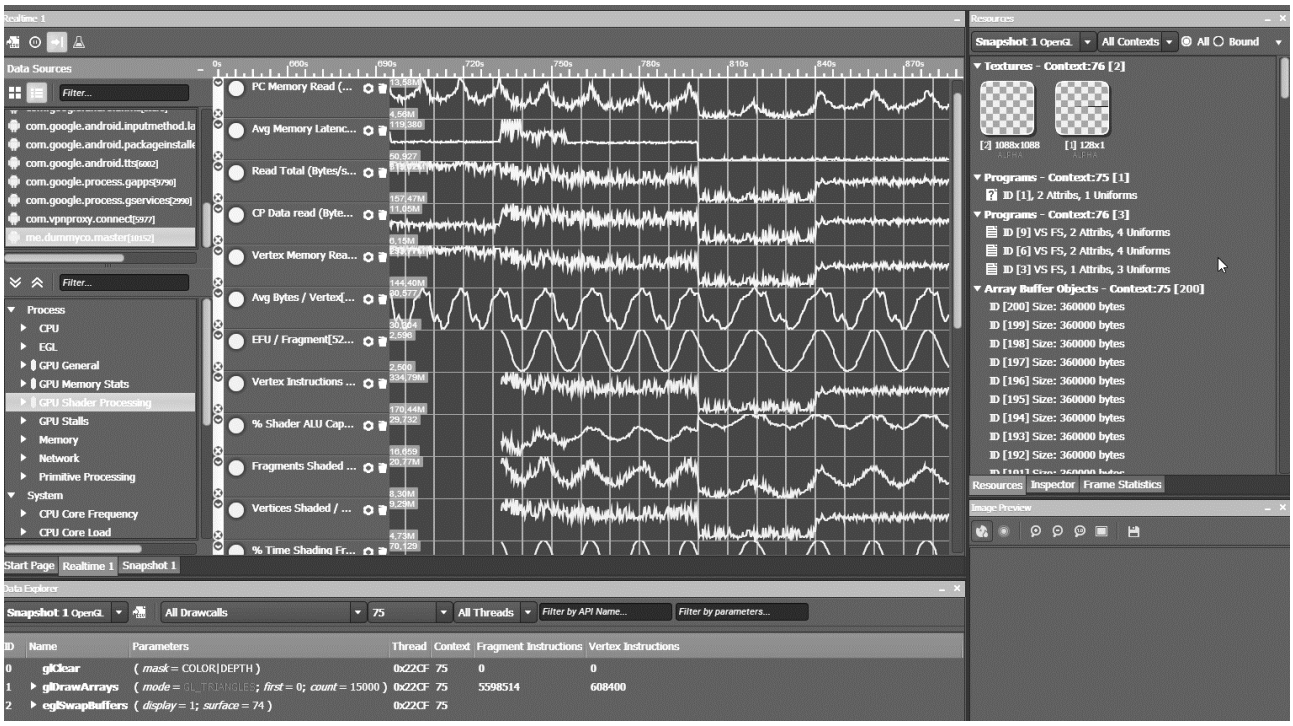


Рисунок 3.6 — Snapdragon Profiler в ході профілювання

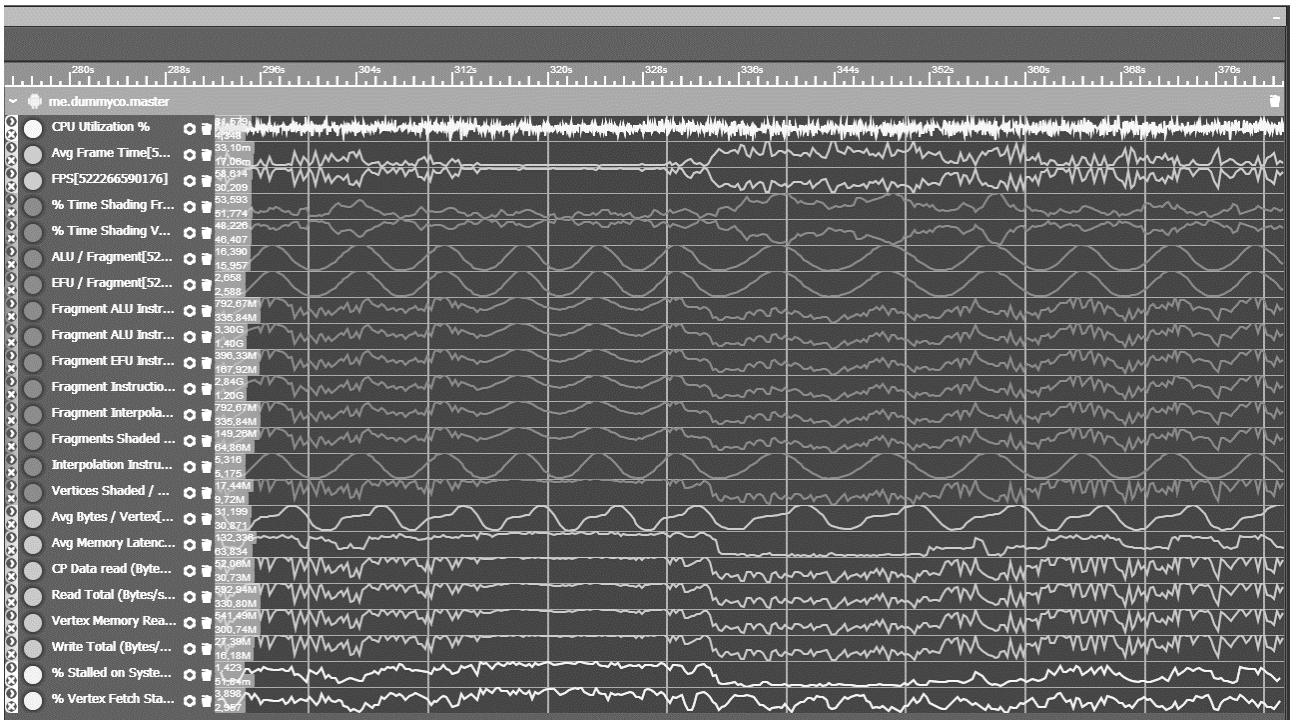


Рисунок 3.7 — Snapdragon Profiler в ході профілювання

3.3 Вершинний буфер, VBO, Розміри пакетів примітивів

VBO — одна з головних особливостей OpenGL та обраного стеку тривимірного рендерингу, навколо котрої будується основна частина дослідження. Одним із факторів досягнення високої ефективності, як вже було зазначено в попередніх розділах — мінімізація кількості змін станів конвеєру на ряду з імплементацією і використанням технік поточного оновлення буферів. Інакше кажучи, необхідно досягти максимальної кількості вершинних даних в рамках одного буферу між змінами прив’язаних програм-шейдерів, прив’язаних текстур, uniform-змінних і т.д. Більш того, інтерфейс OpenGL ES має, звісно, нативну реалізацію з використанням Java Native Interface (далі — JNI), що, в свою чергу, хоч і дає приріст ефективності виконання нативного коду, проте також створює вузьке місце в продуктивності за рахунок виникнення накладних витрат виклику цих самих нативних функцій.

Та ж проблема актуальна і у контексті обробки вершинних даних моделей та матриць афінних перетворень перед і під час загрузки у вершинні буфери, адже, як буде розглянуто у подальших підрозділах, максимальна ефективність обробки вершинних даних та матриць афінних перетворень досягається при їх пакетному постачанні.

Виходячи з цих міркувань, необхідно визначити та імплементувати окрім іншого таку конфігурацію розмірності і кількості буферів вершин, щоб по можливості відв’язатися від повної залежності часу рендерингу від процесорного часу. При цьому підході, центральний процесор повинен звільнитися від виконання будь-яких задач рендерингу до того, як рендеринг на стороні конвеєра закінчиться. Таким чином можливо буде зменшити загальне навантаження на центральний процесор, звільнити процесорний час для виконання інших задач або підготовки наступних заходів рендерингу, збільшити корисне навантаження на графічний процесор та ефективність використання оперативної пам’яті в частині швидкості і кількості оброблених графічним процесором даних.

Для знаходження рішення даної проблеми був частково імплементований, використаний та конфігурований досліджений графічний стек. В частині питання вершинних буферів, їх кількості, розмірності та зняття лімітів максимальної потужності рендерингу зі сторони центрального процесора був проведений ряд бенчмарків різноманітних конфігурацій вершинних буферів для рендерингу попередньо оглянутої моделі, що складається с 871414 примітивів, а саме з 3 вершин на кожен примітив, 3 компонент позиції і 3 компонент вектору нормалі до поверхні примітиву.

На рисунках 3.8 та 3.9 можна спостерігати етап зневадження рендерингу одного вершинного буфера на 500000 примітивів з різних ракурсів. На рисунку 3.10 можна спостерігати порядок деталізації моделі в рамках одного буфера:

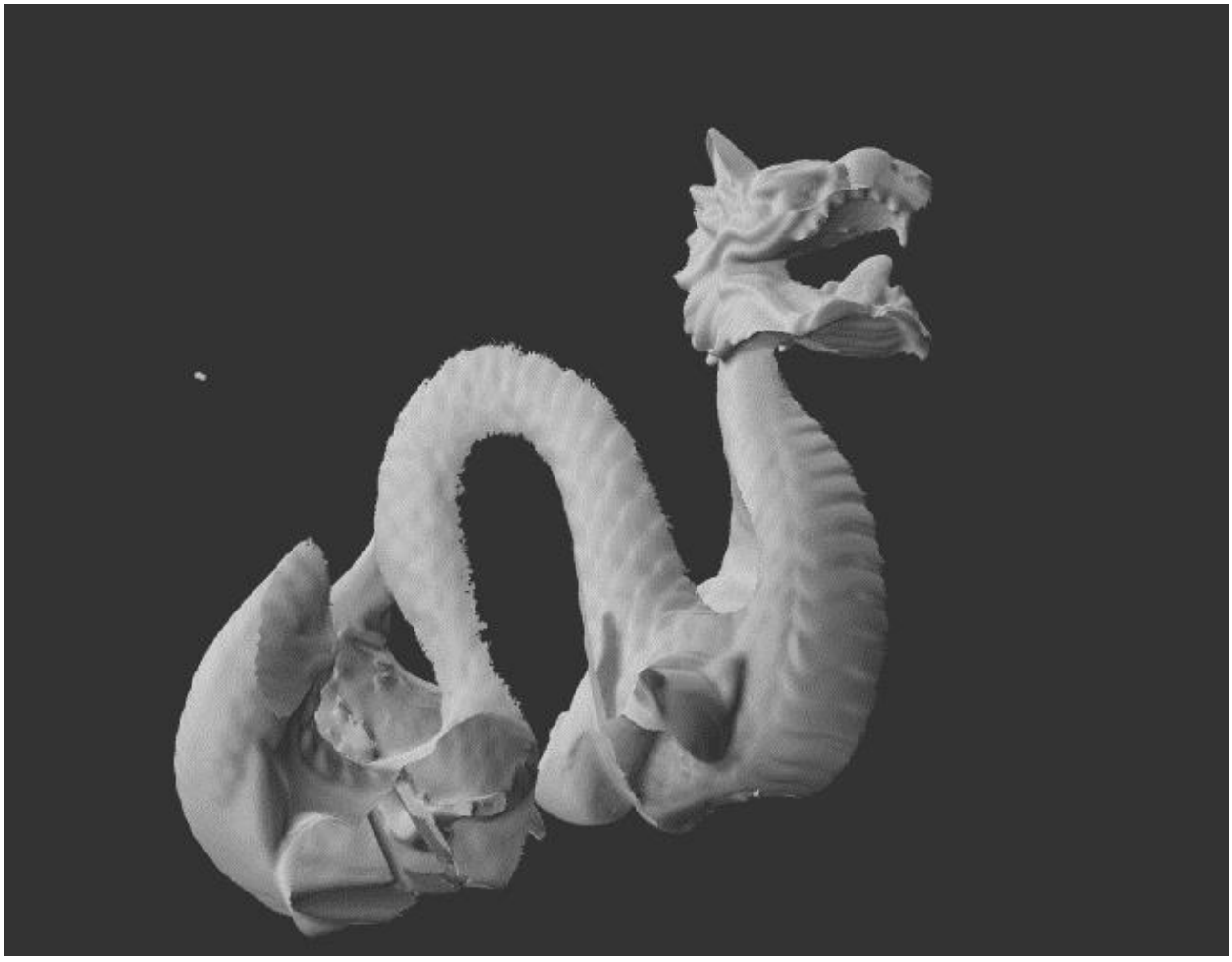


Рисунок 3.8 — Зневадження рендерингу одного вершинного буфера

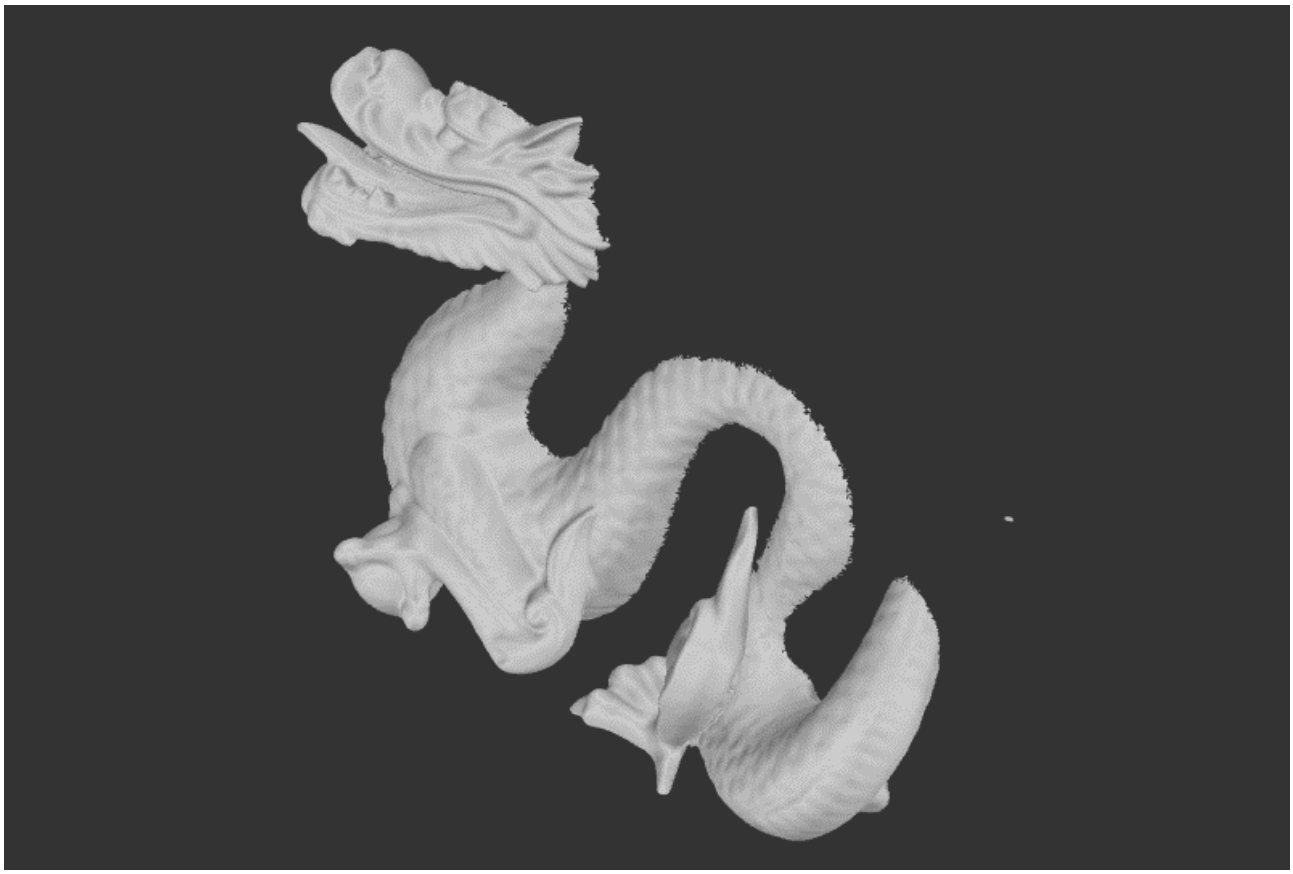


Рисунок 3.9 — Зневадження рендерингу одного вершинного буфера

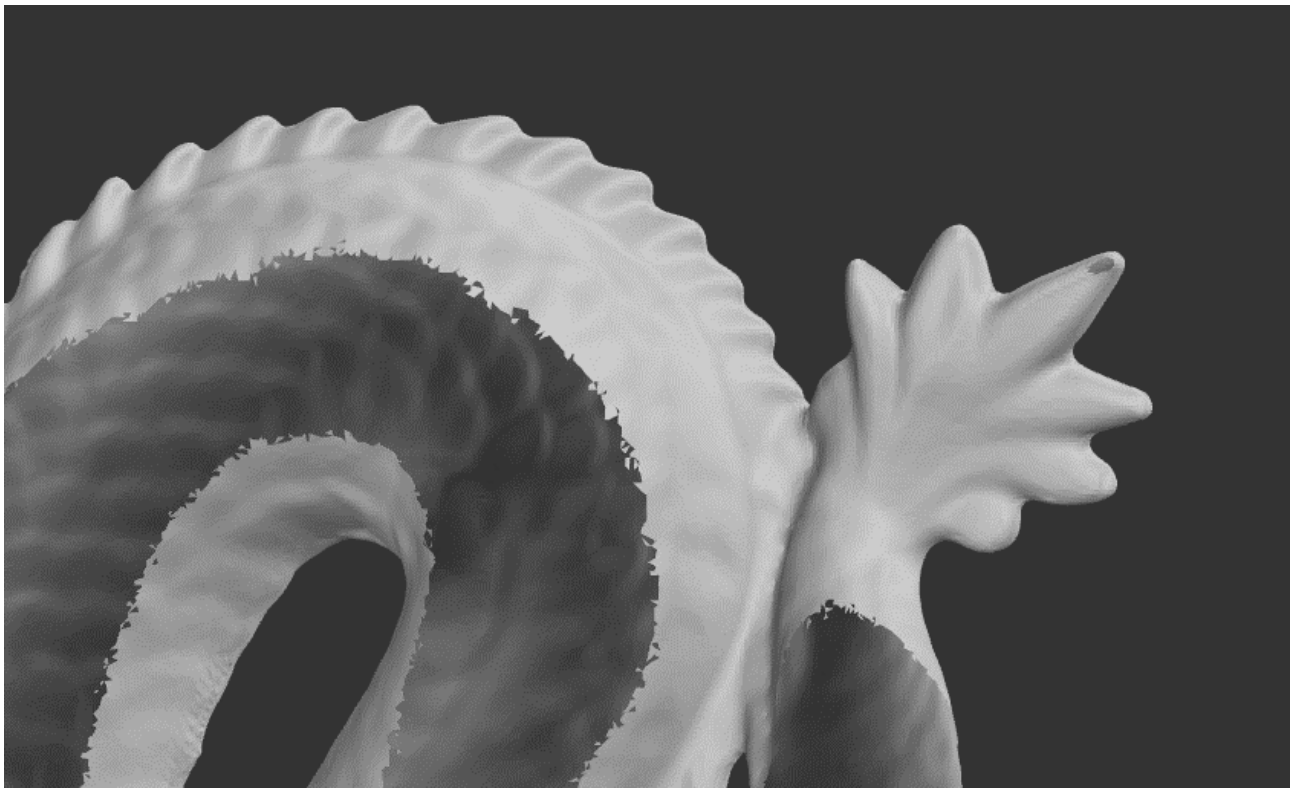


Рисунок 3.10 — Порядок деталізації моделі в рамках одного буфера

На рисунку 3.11 представлений знімок екрана в «бойових» умовах профілювання рішення в інтерфейсі Android пристрою Google Pixel XL [48]:

Рисунок 3.11 — Знімок екрана в «бойових» умовах

В результаті детального профілювання рендерингу 871414 примітивів пакетами з кількістю примітивів у 200, 250, 500, 1000, 2500, 5000, 10000, 20000, 50000, 100000, 150000, 200000, 300000, 350000, 400000, 450000, 500000 одиниць були отримані результати по ряду метрик, представлені на рисунках 3.14 та 3.15.

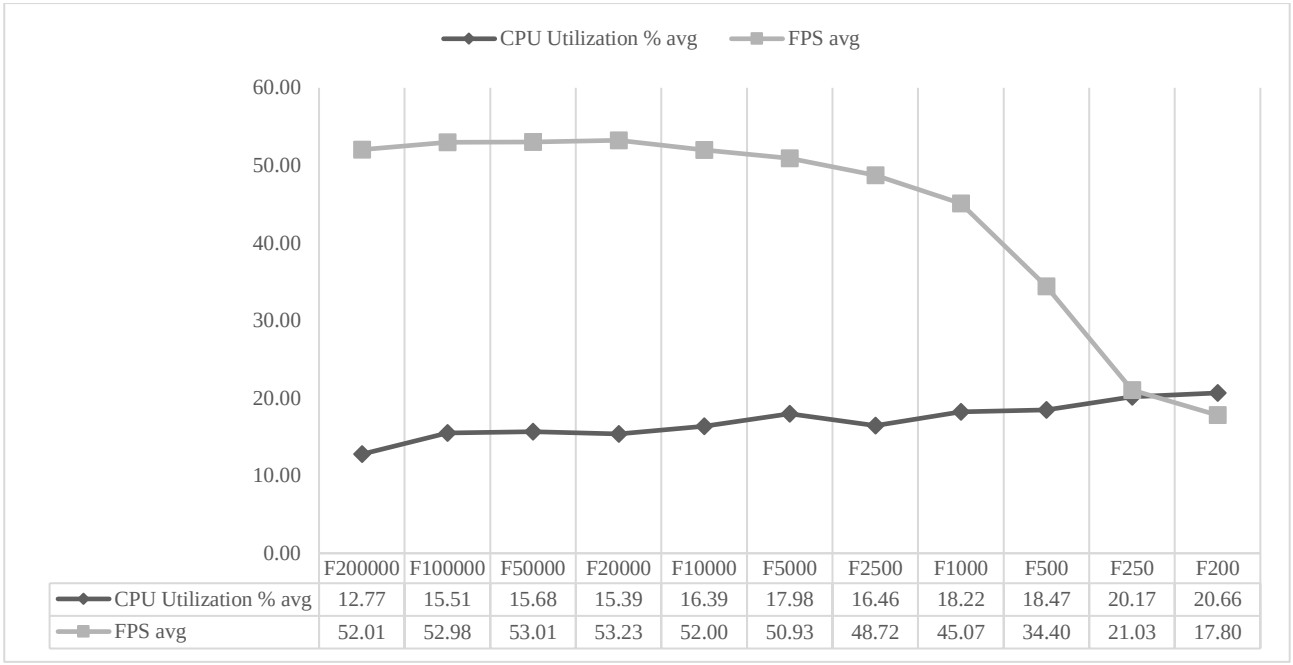
Результати були розбиті на дві групи метрик відповідно до порядків їх значень. Далі буде приведений опис використаних метрик:

- % Shaders Busy avg — середнє значення зайнятості програм-шейдерів
- % Stalled on System Memory avg — середнє значення відносної кількості часу, приділеного для операцій з пам’яттю
- % Vertex Fetch Stall avg — середнє значення відносної кількості часу, приділеного для отримання вершинних даних
- CPU Utilization % avg — середнє значення зайнятості центрального процесору
- FPS avg — середнє значення кількості відмальованих кадрів в секунду
- Fragment ALU Instructions / Sec (Full) — середня кількість фрагментних інструкцій арифметично-логічного пристрою над числами одинарної точності за секунду
- Fragment ALU Instructions / Sec (Half) — середня кількість фрагментних інструкцій арифметично-логічного пристрою над числами половинної точності за секунду
- Fragment EFU Instructions / Second avg — середня кількість фрагментних інструкцій елементарних функцій за секунду
- Fragment Instructions / Second avg — середня кількість фрагментних інструкцій за секунду
- Fragments Shaded / Second avg — середня кількість оброблених фрагментів за секунду
- Read Total (Bytes/sec) avg — середня загальна кількість прочитаних байт за секунду

- Vertex Instructions / Second avg — середня кількість вершинних інструкцій за секунду
- Vertex Memory Read (Bytes/Second) avg — середня кількість прочитаних байт вершинних даних за секунду
- Vertices Shaded / Second avg — середня кількість оброблених вершин за секунду

Слід зазначити, шаг збільшення розмірності буферу змінний і набуває постійного значення у 50000 лише на проміжку 50000–500000 примітивів (рисунок 3.16).

Отримані дані можна також поділити на дві підгрупи: група, зав’язана на потужності центрального процесору та група зав’язана на потужності графічного процесору. Спостерігається різке зниження темпу росту продуктивності рендерингу з ростом розмірності вершинного буферу починаючи з розмірності у 10000 примітивів включно при зниженні завантаженості центрального процесору. Це спостереження також підтверджується профілюванням продуктивності рендерингу подібної моделі на 100000 примітивів (рисунок 3.12):

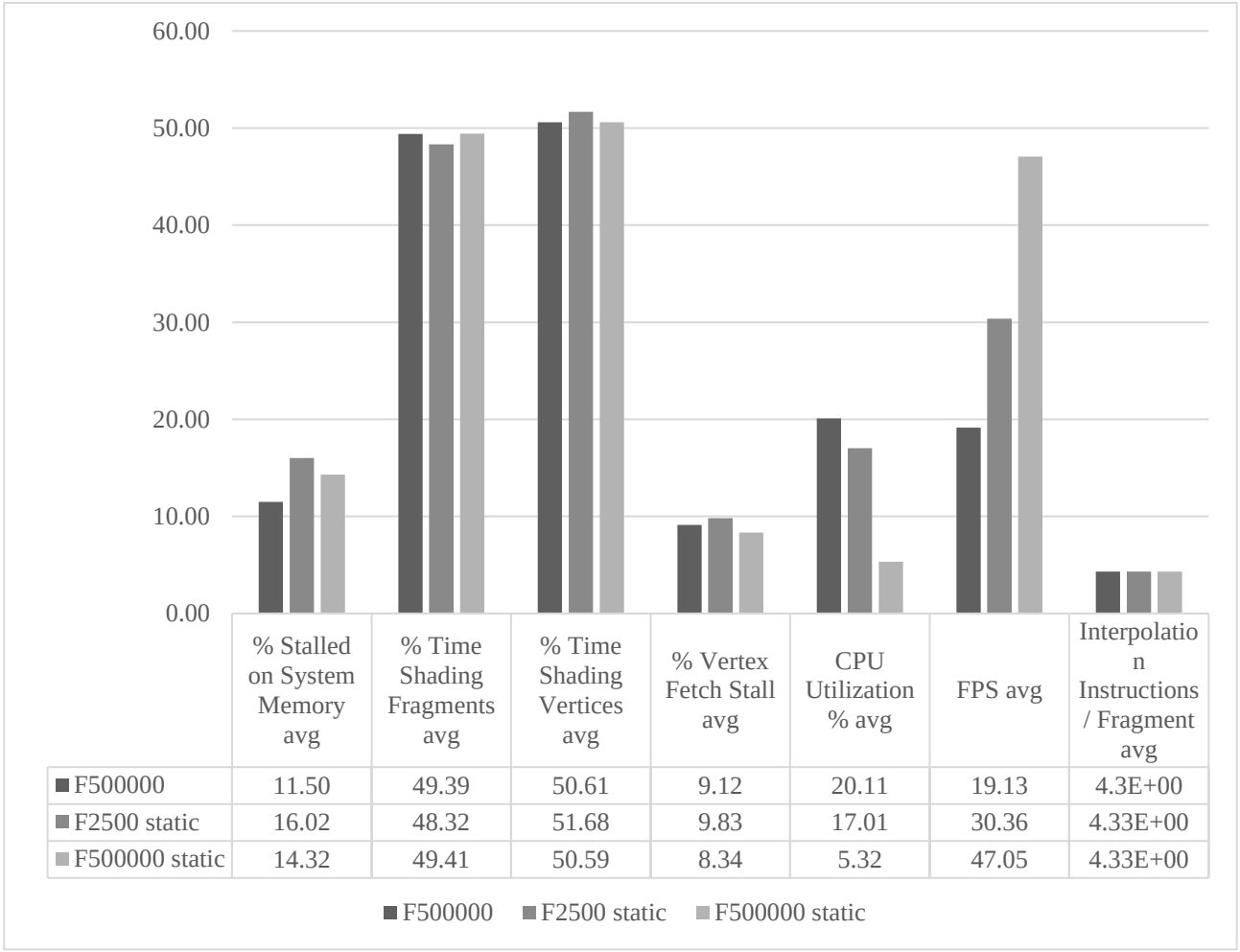


	F200000	F100000	F50000	F20000	F10000	F5000	F2500	F1000	F500	F250	F200
CPU Utilization % avg	12.77	15.51	15.68	15.39	16.39	17.98	16.46	18.22	18.47	20.17	20.66
FPS avg	52.01	52.98	53.01	53.23	52.00	50.93	48.72	45.07	34.40	21.03	17.80

Рисунок 3.12 — Результати профілювання

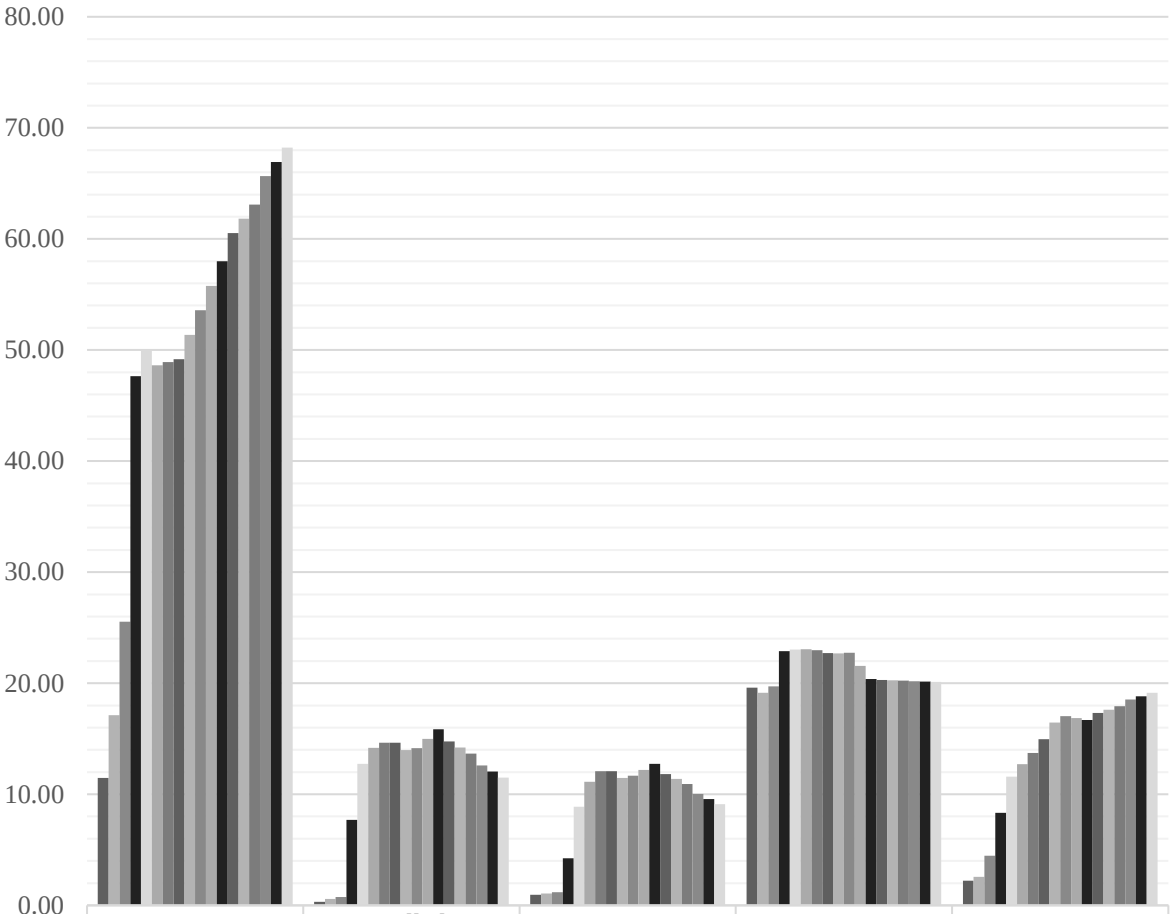
Отримані дані породжують наступні висновки:

1) Зняття лімітів максимальної потужності рендерингу зі сторони центрального процесора критичне і досягається за рахунок мінімізації кількості змін стану графічного конвеєру та оптимізації процедур пов’язаних з обробкою, зберіганням та пакуванням вершинних даних.
2) Збільшення розміру пакету даних вершинного буфера призводить до покращення показників продуктивності рендерингу не тільки за рахунок відповідного зменшення накладних затрат, описаних у попередньому пункті, а й за рахунок внутрішньої реалізації конвеєру, як це видно з рисунка 3.13, на якому представлені результати профілювання продуктивності рендерингу змінними пакетами по 500000 примітивів та статичними пакетами по 2500 і 500000 примітивів.



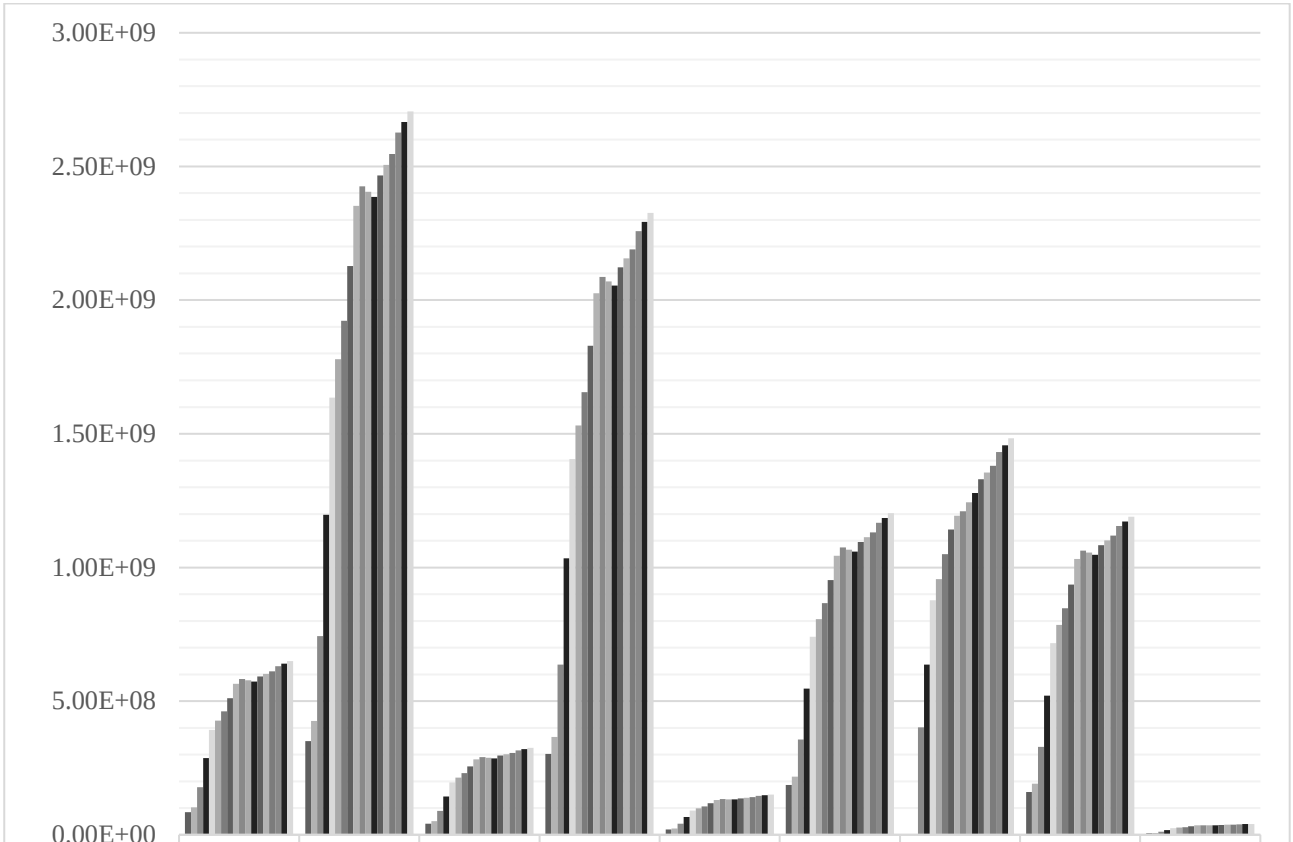
	% Stalled on System Memory avg	% Time Shading Fragments avg	% Time Shading Vertices avg	% Vertex Fetch Stall avg	CPU Utilization % avg	FPS avg	Interpolation Instructions / Fragment avg
F500000	11.50	49.39	50.61	9.12	20.11	19.13	4.3E+00
F2500 static	16.02	48.32	51.68	9.83	17.01	30.36	4.33E+00
F500000 static	14.32	49.41	50.59	8.34	5.32	47.05	4.33E+00

Рисунок 3.13 — Результати профілювання



	% Shaders Busy avg	% Stalled on System Memory avg	% Vertex Fetch Stall avg	CPU Utilization % avg	FPS avg
F200	11.47	0.31	0.96	19.59	2.23
F250	17.11	0.56	1.06	19.14	2.57
F500	25.54	0.74	1.18	19.70	4.46
F1000	47.65	7.71	4.24	22.89	8.33
F2500	50.07	12.74	8.88	23.03	11.59
F5000	48.64	14.18	11.13	23.06	12.71
F10000	48.90	14.64	12.08	22.97	13.72
F20000	49.17	14.65	12.07	22.72	14.95
F50000	51.37	13.97	11.49	22.69	16.45
F100000	53.57	14.14	11.66	22.75	17.03
F150000	55.77	15.00	12.20	21.56	16.86
F200000	57.98	15.85	12.73	20.37	16.70
F250000	60.54	14.76	11.83	20.30	17.31
F300000	61.82	14.22	11.38	20.27	17.61
F350000	63.10	13.68	10.93	20.24	17.92
F400000	65.66	12.59	10.02	20.17	18.53
F450000	66.94	12.04	9.57	20.14	18.83
F500000	68.22	11.50	9.12	20.11	19.13

Рисунок 3.14 — Результати профілювання



	Fragment ALU Instructions / Sec (Full) avg	Fragment ALU Instructions / Sec (Half) avg	Fragment EFU Instructions / Second avg	Fragment Instructions / Second avg	Fragments Shaded / Second avg	Read Total (Bytes/sec) avg	Vertex Instructions / Second avg	Vertex Memory Read (Bytes/Second) avg	Vertices Shaded / Second avg
F200	8.4E+07	3.5E+08	4.2E+07	3.0E+08	2.0E+07	1.9E+08		1.6E+08	5.4E+06
F250	1.0E+08	4.3E+08	5.1E+07	3.7E+08	2.4E+07	2.2E+08		1.9E+08	6.4E+06
F500	1.8E+08	7.4E+08	8.9E+07	6.4E+08	4.1E+07	3.6E+08	4.0E+08	3.3E+08	1.1E+07
F1000	2.9E+08	1.2E+09	1.4E+08	1.0E+09	6.6E+07	5.5E+08	6.4E+08	5.2E+08	1.8E+07
F2500	3.9E+08	1.6E+09	2.0E+08	1.4E+09	9.0E+07	7.4E+08	8.8E+08	7.2E+08	2.4E+07
F5000	4.3E+08	1.8E+09	2.1E+08	1.5E+09	9.9E+07	8.1E+08	9.6E+08	7.9E+08	2.7E+07
F10000	4.6E+08	1.9E+09	2.3E+08	1.7E+09	1.1E+08	8.7E+08	1.0E+09	8.5E+08	2.9E+07
F20000	5.1E+08	2.1E+09	2.6E+08	1.8E+09	1.2E+08	9.5E+08	1.1E+09	9.4E+08	3.2E+07
F50000	5.6E+08	2.4E+09	2.8E+08	2.0E+09	1.3E+08	1.0E+09	1.2E+09	1.0E+09	3.5E+07
F100000	5.8E+08	2.4E+09	2.9E+08	2.1E+09	1.3E+08	1.1E+09	1.2E+09	1.1E+09	3.6E+07
F150000	5.8E+08	2.4E+09	2.9E+08	2.1E+09	1.3E+08	1.1E+09	1.2E+09	1.1E+09	3.6E+07
F200000	5.7E+08	2.4E+09	2.9E+08	2.1E+09	1.3E+08	1.1E+09	1.3E+09	1.0E+09	3.5E+07
F250000	5.9E+08	2.5E+09	3.0E+08	2.1E+09	1.4E+08	1.1E+09	1.3E+09	1.1E+09	3.7E+07
F300000	6.0E+08	2.5E+09	3.0E+08	2.2E+09	1.4E+08	1.1E+09	1.4E+09	1.1E+09	3.7E+07
F350000	6.1E+08	2.5E+09	3.1E+08	2.2E+09	1.4E+08	1.1E+09	1.4E+09	1.1E+09	3.8E+07
F400000	6.3E+08	2.6E+09	3.2E+08	2.3E+09	1.5E+08	1.2E+09	1.4E+09	1.2E+09	3.9E+07
F450000	6.4E+08	2.7E+09	3.2E+08	2.3E+09	1.5E+08	1.2E+09	1.5E+09	1.2E+09	4.0E+07
F500000	6.5E+08	2.7E+09	3.2E+08	2.3E+09	1.5E+08	1.2E+09	1.5E+09	1.2E+09	4.0E+07

Рисунок 3.15 — Результати профілювання

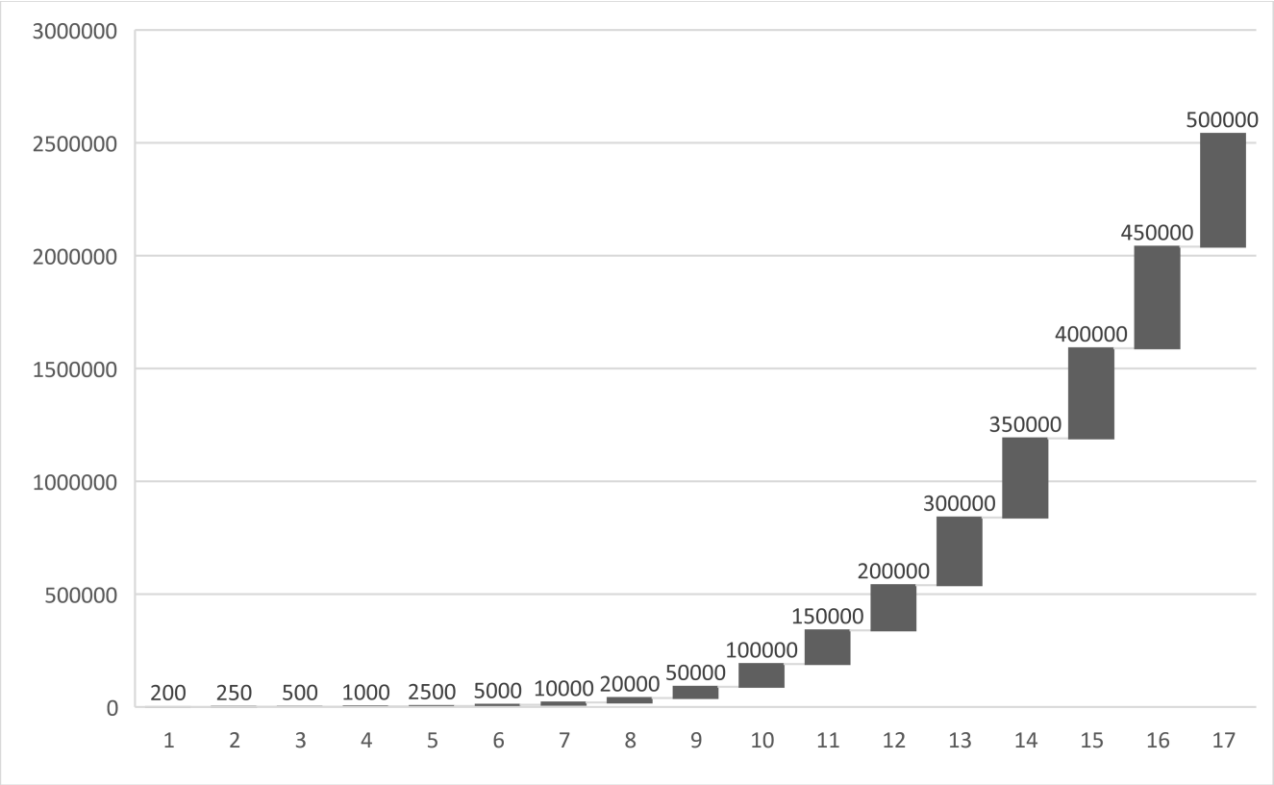


Рисунок 3.16 — Каскадна діаграма кроку кількості примітивів у профільованих вершинних буферах

3.4 Індексний буфер, IBO

Завдяки використанню індексного буфера при рендерингу тривимірною графіки стає можливим відчутне скорочення об’єму даних для передачі і обробки завдяки виключенню з них дублікатів інформації про вершини, що спільно використовуються декількома примітивами, як це було детально розглянуто в підрозділах 1.5.2.1 та 2.5. Розглянемо переваги використання цього підходу на прикладі рендерингу ікосфери — способу апроксимації сфери симпліціальним багатогранником, що формується способом розбиття на трикутні примітиви багатогранника Гольдберга (рисунок 3.17):

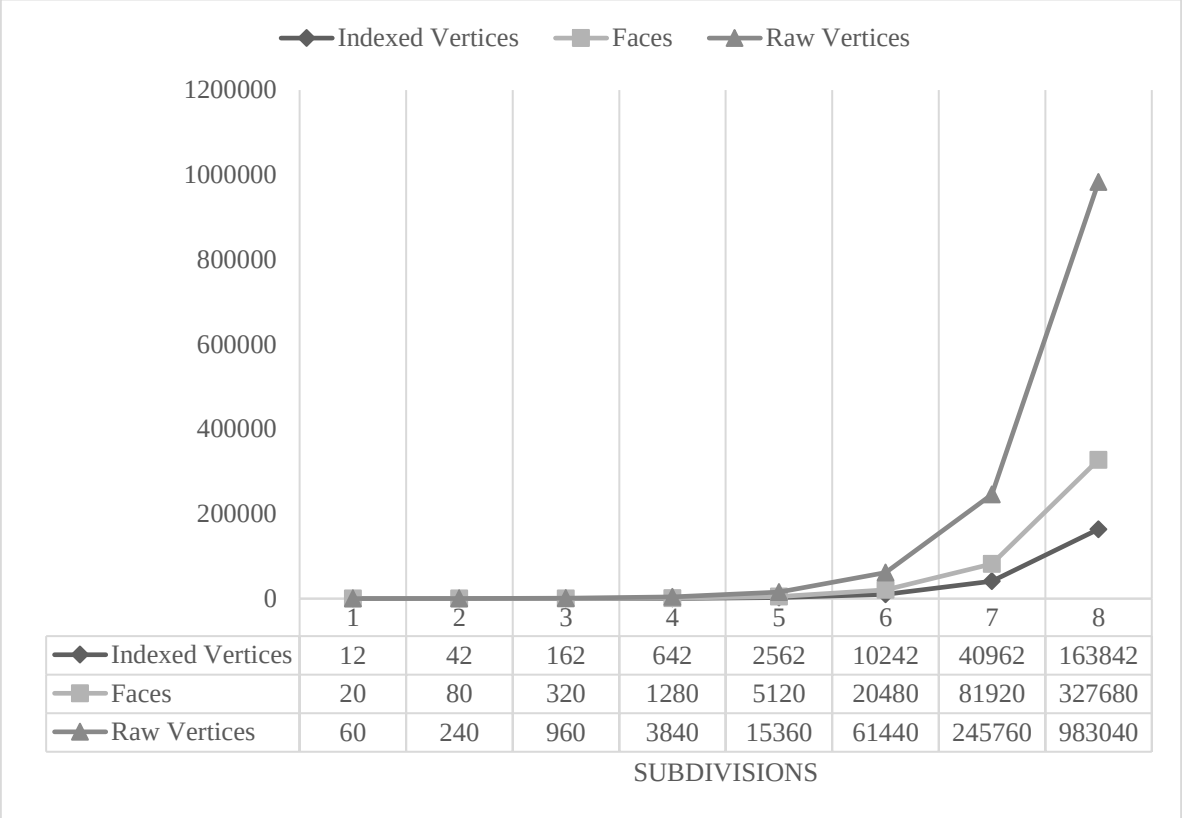


Рисунок 3.17 — Графік залежності кількості індексованих вершин, примітивів та вихідних вершин від рівню розбиття ікосфери, де Indexed Vertices — кількість унікальних вершин, Faces — кількість примітивів, Raw Vertices — загальна кількість вершин, Subdivisions — ступінь розбиття

Як видно з графіку, в даному прикладі можливо скоротити об’єм вершинних даних для передачі майже в 6 разів отримавши лише три додаткових індекси розміром в 2–4 байти на кожен примітив. Зрозуміло, що зі збільшенням кількості даних на кожну вершину, індексний буфер буде давати вагомі переваги. Проте, слід також відмітити, що зазвичай кожна вершина може містити дані, призначені лише для примітиву, в котрому дана вершина задіяна. Наприклад, повершинні значення нормалей до поверхні, значення кольору, значення текстурних координат і т.д. Відповідно за наявністю таких даних використання індексного буферу стає занадто комплексним або неможливим.

3.5 Vertex Array Object, VAO

Vertex Array Object, VAO — об'єкт OpenGL, який зберігає весь стан, необхідний для передачі вершинних даних. Він зберігає формат даних вершин і стан буферних об'єктів, що забезпечують вершинні масиви даних. VAO не копіює, не закріплює та не зберігає вміст згаданих буферів — лише посилається на них. При зміні будь-яких даних у відповідних буферах, на які посилається існуючий VAO, ці зміни будуть видно користувачам цього VAO. [49]

Нижче зображені результати профілювання рішення з використанням VAO на основі вершинних буферів розмірністю 500000 (рисунок 3.18):

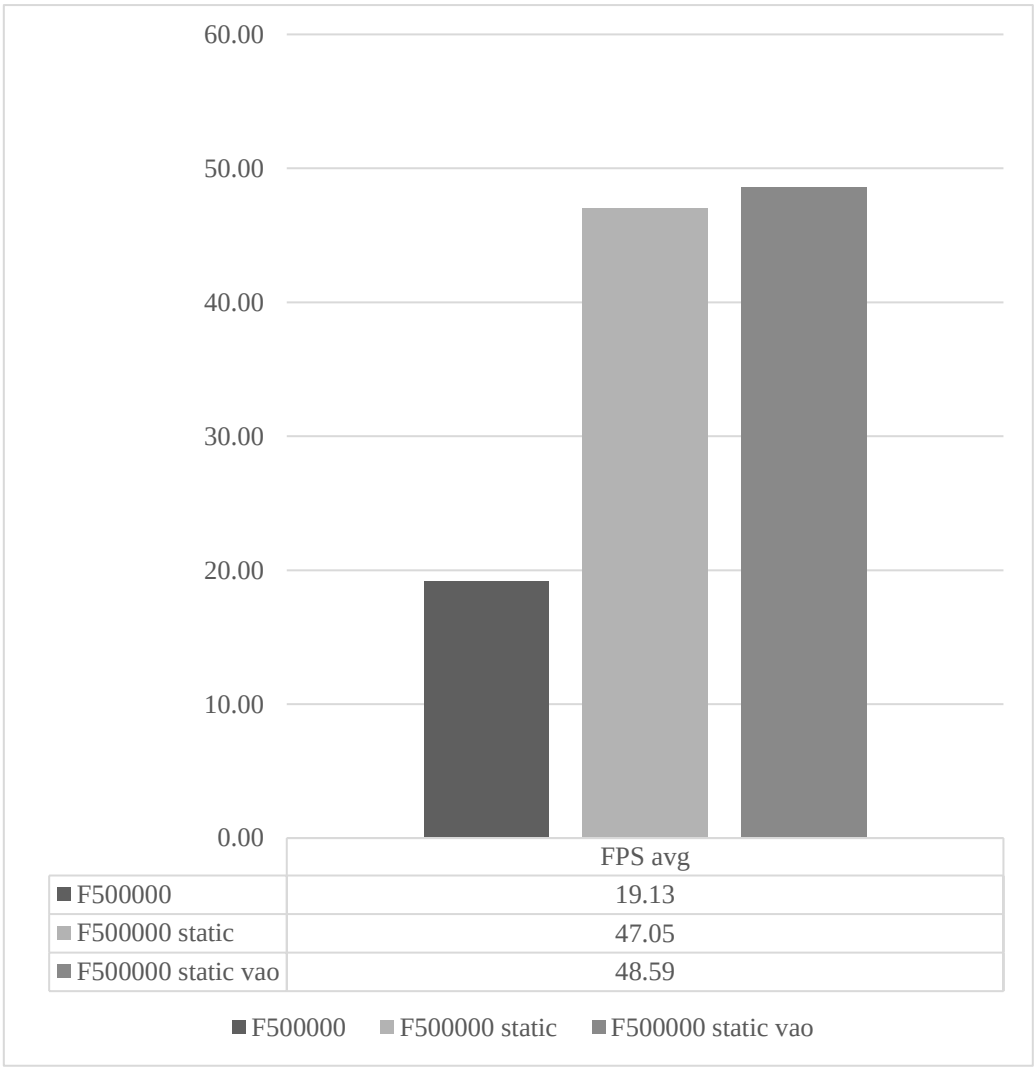


Рисунок 3.18 — Результати профілювання рішення з використанням VAO

3.6 Передача даних у нативні буфери

OpenGL ES API в Android працює з підкласами Buffer [50] для передачі даних у конвеєр. Buffer представляє собою контейнер для даних конкретного примітивного типу. [51]

Buffer — лінійна, кінцева послідовність елементів певного примітивного типу. Окрім його вмісту, основними властивостями буфера є його ємність, ліміт та розташування:

Ємність буфера — це кількість елементів, які він містить. Ємність буфера ніколи не є негативною і ніколи не змінюється.

Ліміт буфера — це індекс першого елемента, який не слід зчитувати чи записувати. Ліміт буфера ніколи не є негативним і ніколи не перевищує його ємність.

Позиція буфера — це індекс наступного елемента для читання чи запису. Позиція буфера ніколи не є негативною і ніколи не перевищує його ємності.

Існує один підклас даного класу для кожного не логічного примітивного типу.

Байтовий буфер може бути прямий або непрямий. Отримавши прямий буфер байтів, віртуальна машина Java докладе максимум зусиль для безпосереднього виконання власних операцій вводу-виводу. Тобто вона намагатиметься уникати копіювання вмісту буфера до (або від) проміжного буфера до (або після) кожного виклику однієї з базових операцій нативного вводу-виводу.

Прямий байтовий буфер може бути створений за допомогою фабричного методу allocateDirect() цього класу. Буфери, що повертаються за допомогою цього методу, зазвичай мають дещо більш накладні витрати на виділення та звільнення, ніж непрямі буфери. Вміст прямих буферів може перебувати за межами звичайної кучі, і тому їх вплив на обсяг пам'яті програми може бути неочевидним. Тому рекомендується, щоб прямі буфери виділялися насамперед для великих, довгоживучих буферів, які підлягають операціям вводу-виводу

основної системи. Загалом найкраще виділяти прямі буфери лише тоді, коли вони приносять помітний приріст у продуктивності програми.

OpenGL ES API в Android працює з прямими байтовими буферами.

Ефективне заповнення буферів вершинними даними — також один із основних факторів високої потужності рішення. В рамках даної платформи та стеку технологій існують наступні способи вирішення цієї задачі:

- Використання проміжного масиву для накопичення вершинних даних засобами System.arraycopy() та одноразового копіювання в буфер пропрієтарними нативними засобами.
- Використання проміжного масиву для накопичення вершинних даних та одноразового копіювання в буфер засобами Android SDK, NDK та JDK
- Використання нативного пропрієтарного підходу заповнення буферу попримітивно
- Використання стандартних засобів Android SDK, NDK та JDK для заповнення буферів попримітивно

Результати профілювання кожного розглянутого підходу (рисунок 3.19):

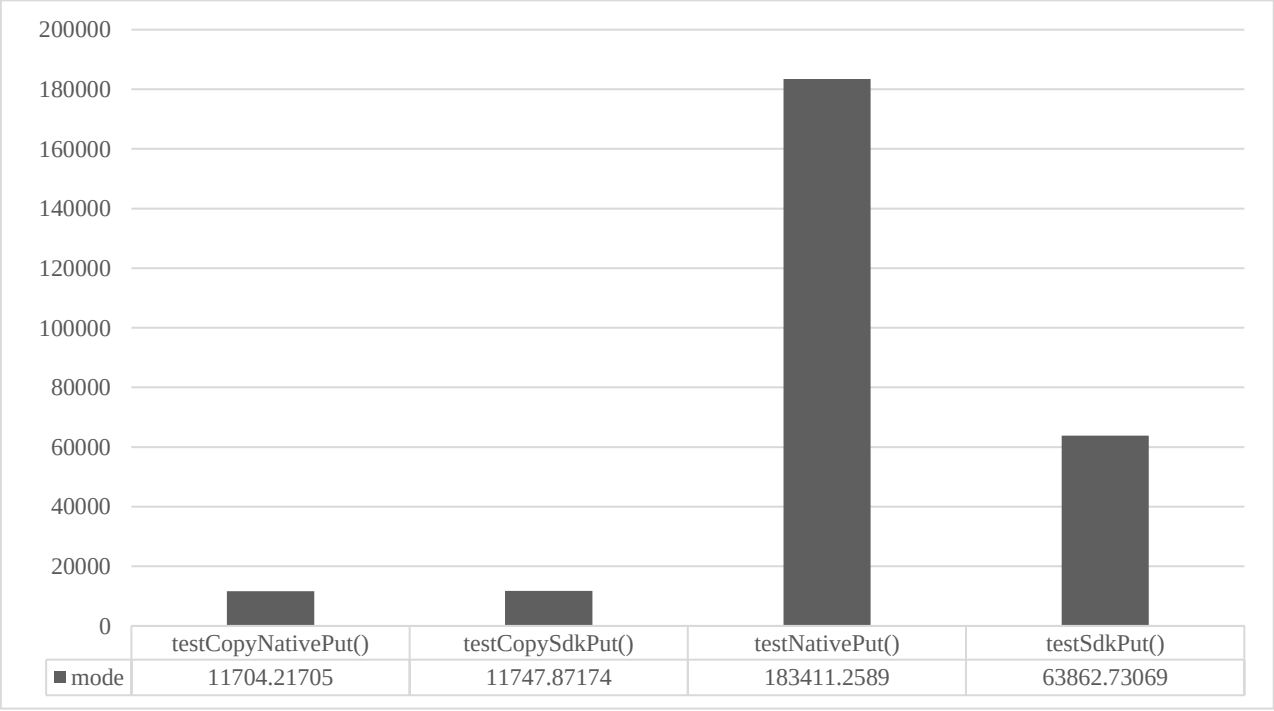


Рисунок 3.19 — Результати профілювання

Тести на представленій діаграмі відповідають їх порядку у списку вище. По вертикалі представлений час, затрачений на виконання 100 тестових проходів заповнення буферу на 9000000 елементів з рухомою комою у мілісекундах. На наступному рисунку представлений приклад пропріетарних нативних методів заповнення буферів (рисунок 3.20):

```
extern "C"
JNIEXPORT void JNICALL
Java_me_dummyco_master_util_BufferUtils_putFloatsJni(JNIEnv *env, jclass type, jfloatArray src_,
                                                     jobject obj_dst, jint numFloats,
                                                     jint srcOffset) {
    unsigned char *dst = (unsigned char *) (obj_dst ? env->GetDirectBufferAddress(obj_dst) : 0);
    jfloat *src = env->GetFloatArrayElements(src_, NULL);

    memcpy(dst, src + srcOffset, static_cast<size_t>(numFloats << 2));

    env->ReleaseFloatArrayElements(src_, src, 0);
}

extern "C"
JNIEXPORT void JNICALL
Java_me_dummyco_master_util_BufferUtils_putShortsJni(JNIEnv *env, jclass type, jshortArray src_,
                                                     jobject obj_dst, jint numShorts,
                                                     jint srcOffset) {
    unsigned char *dst = (unsigned char *) (obj_dst ? env->GetDirectBufferAddress(obj_dst) : 0);
    jshort *src = env->GetShortArrayElements(src_, NULL);

    memcpy(dst, src + srcOffset, static_cast<size_t>(numShorts << 2));

    env->ReleaseShortArrayElements(src_, src, 0);
}
```

Рисунок 3.20 — Приклад пропріетарних нативних методів заповнення буферів

memcpy — функція стандартної бібліотеки C для копіювання вмісту однієї області пам’яті в іншу. Має наступну сигнатуру (3.1):

$$void* \ memcpy(\ void* \ dest, \ const \ void* \ src, \ std::size\_t \ count \ ); \quad (3.1)$$

Копіює *count* байт з об’єкту, вказаного в *src* у об’єкт, на котрий вказує *dest*. Обидва об’єкти сприймаються як масиви *unsigned char*. При накладанні об’єктів поведінка непередбачувана. При нульовому вказівнику у *dest* або *src* поведінка

непередбачувана, навіть с нульовим *count*. memcpy призваний бути найшвидшим бібліотечним методом копіювання вмісту однієї області пам’яті в іншу.

GetDirectBufferAddress отримує та повертає адрес початку області пам’яті, на яку вказує прямий java.nio.Buffer.

Get<PrimitiveType>ArrayElements — сімейство функцій отримання вмісту масиву примітивів. Результати валідні до моменту звільнення відповідного масиву функцією Release<PrimitiveType>ArrayElements(). [52]

System.arraycopy() — метод пакету Java.lang.System, призначений для копіювання масиву примітивів з початкового масиву примітивів зі вказаною початковою позицією до цільового масиву примітивів зі вказаною початковою позицією. [53]

За результатами імплементації та профілювання можна зробити висновок о доцільності використання підходу з використанням проміжного масиву для накопичення вершинних даних засобами System.arraycopy() та одноразового копіювання в буфер пропрієтарними нативними засобами.

3.7 Матриці афінних перетворень

Класична, наївна реалізація добутку матриць передбачає використання стандартних засобів JDK [54] з використанням трьох вкладених циклів.

Для оцінки продуктивності підходу та його наївних аналогів були створені спеціальні програми-бенчмарки з врахуванням всіх «best practices». Існує багато оптимізацій, що можуть бути використані віртуальною Java-машиною або апаратним забезпеченням до конкретного компоненту під час його ізольованого тестування. Непродумані тести продуктивності можуть давати оптимістичні результати, що зовсім відрізняються від реальних. Наприклад, навіть порядок виконання тестів та кількість ітерацій тестування можуть створювати велику похибку в користь певного з підходів, шо тестуються.

Саме тестування конкретного компоненту в складі робочого, практичного додатку с використанням таких технік, як JVM warm up (підігрів віртуальної Java-машини) [55] може дати достовірні результати.

Далі будуть приведені результати тестів продуктивності для ітерацій розробленого підходу та його наївних альтернатив з різним стеком використаних технік та особливостей, їх порівняння. Під метрикою продуктивності мається на увазі час, затрачений процесором на виконання певної кількості матричних добутків.

Результати тестування затрат процесорного часу на добуток 1 мільйону матриць 4×4 у виді одновимірного масиву на 16 елементів типу з рухомою комою за використанням класичної, наївної реалізації засобами JDK на пристрої Google Pixel XL в 10 ітерацій представлені на рисунку 3.20:

Ітерація, №	Затрати процесорного часу, мс
1	3121
2	3141
3	3132
4	3155
5	3137
6	3127
7	3133
8	3115
9	3111
10	3116
Усереднений результат	3129

Рисунок 3.20 — Результати тестування продуктивності наївного підходу

Результати тестів для різної кількості матриць, різної кількості ітерацій, різних пристроїв та різної розмірності матриць опускаються по причині практично прямої залежності між приведеними метриками та продуктивністю обчислень за умови відсутності вузьких місць конкретної платформи для тестування.

Представлені результати отримані на пристрої з процесором Qualcomm MSM8996 Snapdragon 821 Quad-core (2x2.15 GHz Kryo & 2x1.6 GHz Kryo). [56]

Слід відзначити, що тести продуктивності проводились на збірках додатку з використанням агресивної оптимізації та обфускації початково коду за використанням інструменту ProGuard [57] з сімома проходами оптимізації, що, зокрема, застосовують арифметичні оптимізації, фіналізацію класів, їх членів, вертикальне та горизонтальне склеювання класів, приватизацію полів, методів, витягнення статичних методів, оптимізацію кількості параметрів методів, вбудовування константних виразів, коротких та унікальних методів, набір методів «peephole optimization» для арифметичних операцій, гілок коду, строкових даних, приведення типів, чистку надлишкових інструкцій, виключень, блоків коду.

В складі Android SDK присутній набір класів-застосувань для роботи з матрицями. Їх реалізація, подібно й надалі запропонованому підходу, також імплементована на рівні NDK. В цілому, вона повністю дублює зазначений вище підхід (рисунок 3.21). Фрагмент зазначеної реалізації наведений з ядра Android: /base/core/jni/android/opengl/util.cpp за станом на версію Oreo release [58].

Результати тестів продуктивності вбудованого рішення по визначеному алгоритму тестування на ідентичному наборі початкових даних та за ідентичного апаратного та програмного середовища представлений на рисунку 3.21:

Ітерація, №	Затрати процесорного часу, мс
1	6075
2	5997
3	5987
4	6008
5	6037
6	6023
7	6005
8	5994
9	5973
10	6053
Усереднений результат	6015

Рисунок 3.21 — Результати тестування продуктивності нативного вбудованого підходу

Можливо виділити дві основні причини низької продуктивності даної реалізації. Перша – наявність накладних витрат виклику нативних функцій за використанням JNI, так званий «overhead». Друга – наявність цілого ряду додаткових перевірок вхідних даних та надлишкового копіювання даних у тимчасові буфери для проведення обчислень.

Виходячи з проведених тестів, націлених на оцінку витрат виклику JNI за використанням Google Caliper [59] можна зробити висновок, що на цільовому апаратно-програмному забезпеченні виклик JNI функцій може займати в перспективі час еквівалентний 5-30 Java операціям над примітивними типами (рисунок 3.22) [60]:

- Scenario {JniCall}
  10.26 ns; σ=0.02 ns @ 10 trials
- Scenario{AddIntOperation}
  0.48 ns; σ=0.02 ns @ 10 trials
- Scenario{AddLongOperation}
  0.87 ns; σ=0.02 ns @ 10 trials

Тест	Витрати процесорного часу, нс
JniCall	10.265
AddIntOperation	0.481
AddLongOperation	0.873

Рисунок 3.22 — Результати тестування накладних витрат виклику нативних функцій та Java-операцій над примітивними типами

Таким чином, можливо запропонувати вирішення даної задачі прибираючи максимальну кількість надлишкових перевірок і вбудованих методів захисту і перевірки нативної імплементації, що допустимо на такому низькому рівні виконання, оскільки предметна область даного рішення передбачає використання у составі бойових, відлагоджених і стабільних систем, де в принципі не допустима передача не дійсних вхідних параметрів для обчислень, оскільки відповідальність за такі перевірки та препроцесінг в таких системах несуть верхні рівні, абстракції та реалізації стеку рендерингу.

Також можливо оптимізувати представлений наївний алгоритм добутку матриць введенням константних індексів, агресивних рівнів оптимізації компілятора нативного рішення.

Окрім цього, в контексті цих самих систем в більшості випадків існує потреба саме пакетної обробки даних. Маючи можливість отримувати результат добутку десятків, сотень матриць одночасно, можливо, відповідно, багатократно скоротити накладні затрати на виклик високопродуктивних JNI функцій. Цей підхід потребує важкої та високопродуктивної імплементації на рівні клієнтського коду JVM, проте цілком заслуговує його.

Для імплементації запропонованого рішення використовується відносно нова підтримка роботи з NDK в составі середи розробки Android Studio. [61]

```
#define I(_i, _j) ((_j)+ 4*(_i))

static
void multiplyMM(float* r, const float* lhs, const float* rhs)
{
    for (int i=0 ; i<4 ; i++) {
        register const float rhs_i0 = rhs[ I(i,0) ];
        register float ri0 = lhs[ I(0,0) ] * rhs_i0;
        register float ri1 = lhs[ I(0,1) ] * rhs_i0;
        register float ri2 = lhs[ I(0,2) ] * rhs_i0;
        register float ri3 = lhs[ I(0,3) ] * rhs_i0;
        for (int j=1 ; j<4 ; j++) {
            register const float rhs_ij = rhs[ I(i,j) ];
            ri0 += lhs[ I(j,0) ] * rhs_ij;
            ri1 += lhs[ I(j,1) ] * rhs_ij;
            ri2 += lhs[ I(j,2) ] * rhs_ij;
            ri3 += lhs[ I(j,3) ] * rhs_ij;
        }
        r[ I(i,0) ] = ri0;
        r[ I(i,1) ] = ri1;
        r[ I(i,2) ] = ri2;
        r[ I(i,3) ] = ri3;
    }
}

static
void util_multiplyMM(JNIEnv *env, jclass clazz,
    jfloatArray result_ref, jint resultOffset,
    jfloatArray lhs_ref, jint lhsOffset,
    jfloatArray rhs_ref, jint rhsOffset) {

    FloatArrayHelper resultMat(env, result_ref, resultOffset, 16);
    FloatArrayHelper lhs(env, lhs_ref, lhsOffset, 16);
    FloatArrayHelper rhs(env, rhs_ref, rhsOffset, 16);

    bool checkOK = resultMat.check() && lhs.check() && rhs.check();

    if ( !checkOK ) {
        return;
    }

    resultMat.bind();
    lhs.bind();
    rhs.bind();

    multiplyMM(resultMat.mData, lhs.mData, rhs.mData);

    resultMat.commitChanges();
}
```

Рисунок 3.21 — Приклад стандартної реалізації Android

Запропонований підхід має можливість записувати результат добутку не тільки у масив, а й у сімейство класів FloatBuffer [62], що спеціально призначені для впорядкованої організації масивів примітивів у пам’яті для подальшого високопродуктивного доступу інших компонентів та модулів рендерингу до даних в буферах.

В результаті імплементації запропонованих рішень вдалося отримати наступні показники продуктивності для пакетної обробки матриць пакетами по 16 у перерахунку на мільйон матриць (рисунок 3.22):

Ітерація, №	Затрати процесорного часу, мс
1	1056
2	1050
3	1059
4	1053
5	1050
6	1059
7	1065
8	1101
9	1143
10	1086
Усереднений результат	1072

Рисунок 3.22 — Результати тестування продуктивності пропонованого пакетного підходу

Таким чином, запропоноване рішення демонструє результати в середньому у 5,6 разів продуктивніші у порівнянні з нативним рішенням зі складу SDK та в середньому у 2,9 разів продуктивніші у порівнянні з наївним Java-рішенням.

Таких показників вдалося досягнути за рахунок максимального скорочення накладних витрат на багатократний виклик та взаємодію з нативними функціями. Порівняльна візуалізація використаних підходів зображена на рисунку 3.23.

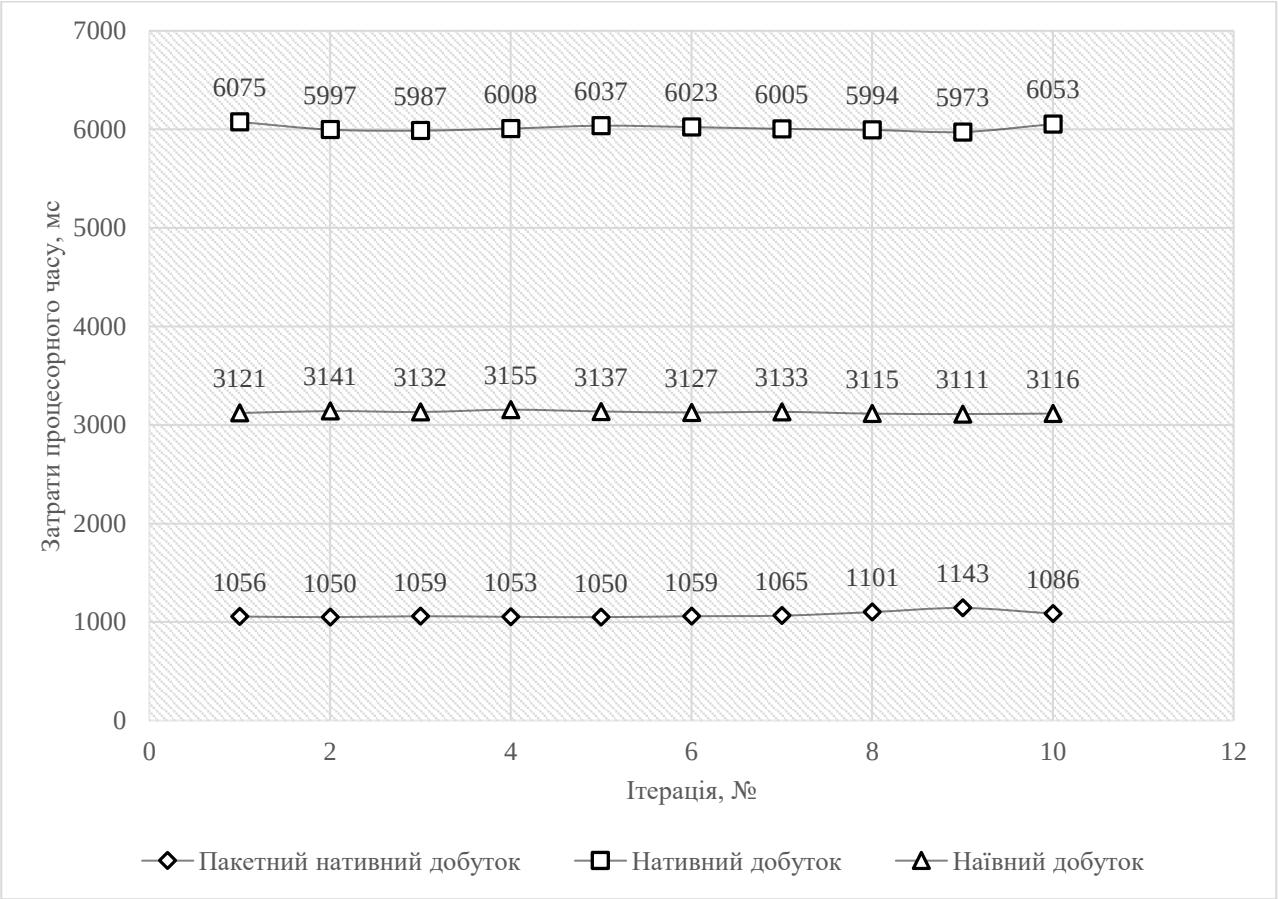


Рисунок 3.23 — Графік залежності затрат процесорного часу від ітерації тестування для всіх розібраних та запропонованих підходів

3.8 Механізм роботи

Таким чином, в результаті імплементації та конфігурації технік, особливостей та підходів, розглянутих у попередніх підрозділах, була додатково імплементована підсистема у складі додатку для їх організації та інкапсуляції використання. Загальна організація модулів підсистеми зображена на рисунку 3.24.

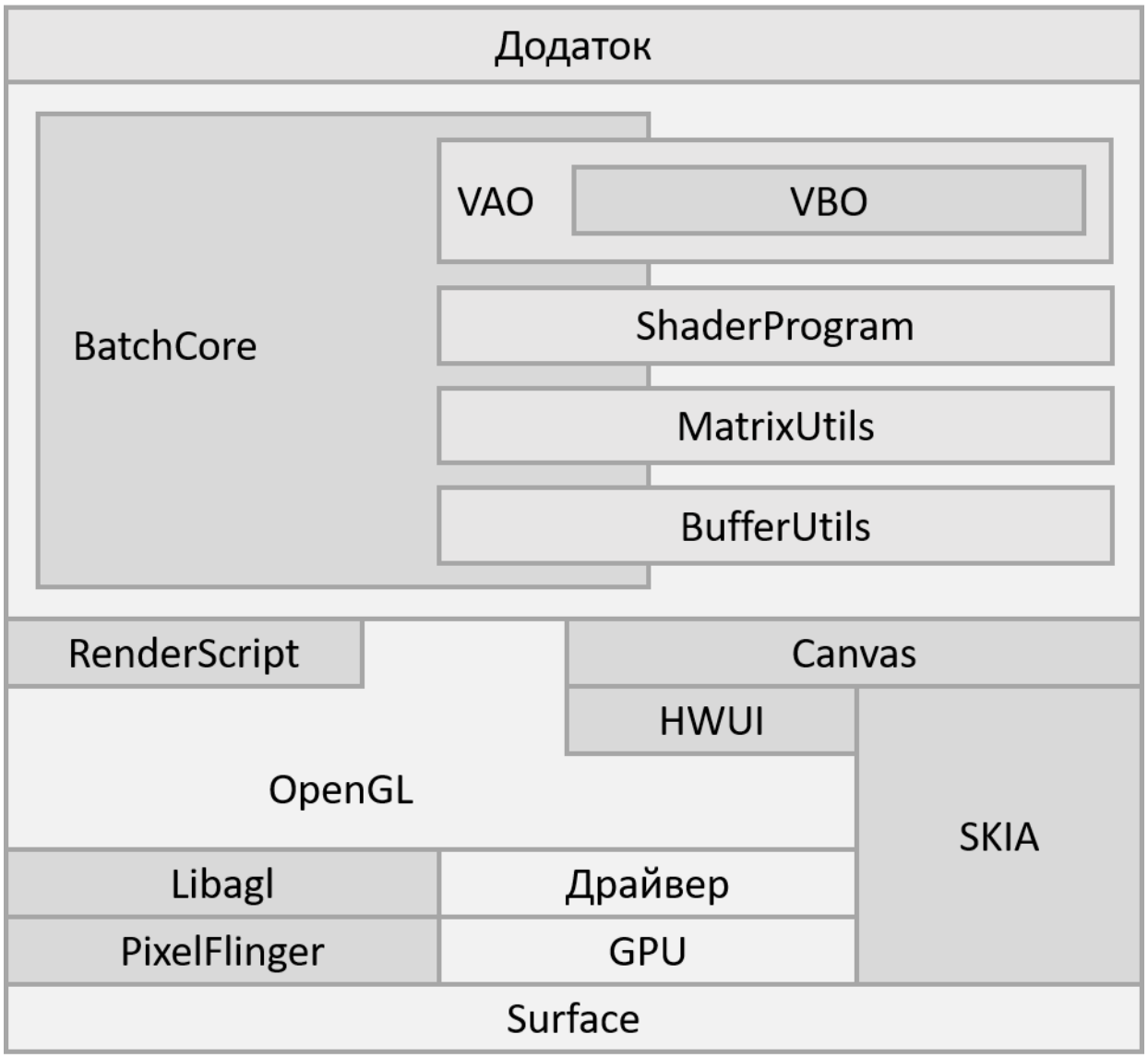


Рисунок 3.24 — Загальна організація модулів підсистеми

BatchCore представляє групу менеджер-класів основної взаємодії з OpenGL ES 3.0 API. Вони надають основні інтерфейси взаємодії додатка з розробленою підсистемою.

VAO представляє групу менеджер-класів, що інкапсулюють клієнтський стан відповідних Vertex Array Object та інтерфейси взаємодії з ними.

VBO представляє групу менеджер-класів, що інкапсулюють клієнтський стан відповідних Vertex Buffer Object та інтерфейси взаємодії з ними. Окрім цього, ці класи відповідальні за менеджмент прямих буферів java.nio.Buffer та проміжних масивів-буферів.

ShaderProgram представляє групу менеджер-класів, що інкапсулюють клієнтський стан відповідних програм-шейдерів та інтерфейси взаємодії з ними. Окрім цього, ці класи відповідальні за створення, компіляцію, звільнення та використання програм-шейдерів.

MatrixUtil представляє групу класів-утиліт, що надають інтерфейси та реалізації розглянутих методів роботи з матрицями афінних перетворень та кінцевими буферами java.nio.Buffer.

BufferUtils представляє групу класів-утиліт, що надають інтерфейси та реалізації розглянутих методів роботи з буферами java.nio.Buffer.

3.9 Google Caliper

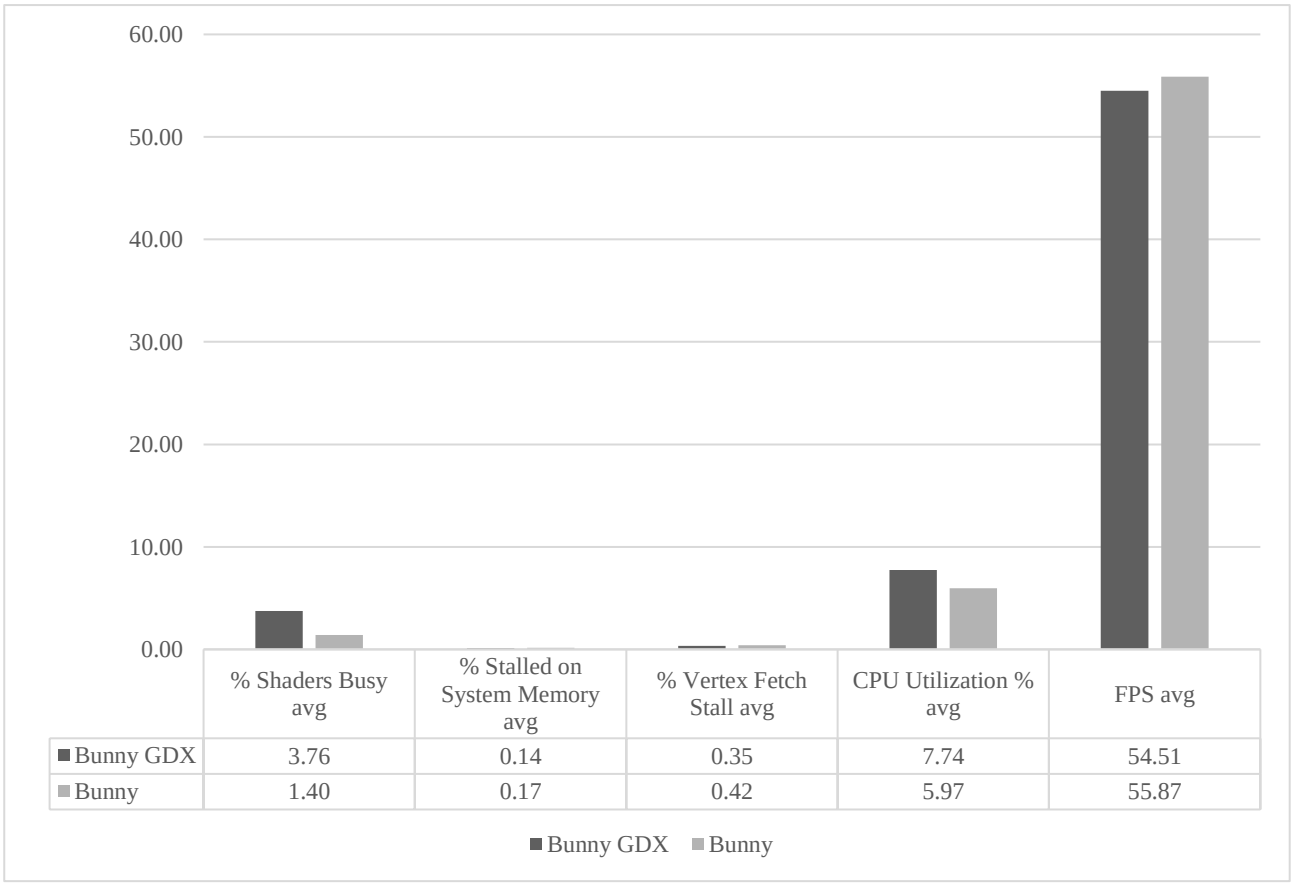
Google Caliper — фреймворк Google із відкритим початковим кодом для написання, запуску та перегляду результатів мікробенчмарків Java. В більшості приведених бенчмарків ефективності імплементацій використаний Google Caliper. Станом на час проводження даного дослідження Google Caliper для Android перебуває в стані активної розробки і не має визначеної документації, підтримки та прикладів застосування. Інструмент був зібраний за доступними на момент дослідження розробками в середовищі Ubuntu [63] (рисунок 3.25):

```
[INFO] Reactor Summary:
[INFO]
[INFO] Caliper Maven Parent ............................... SUCCESS [  1.604 s]
[INFO] Caliper Util ....................................... SUCCESS [  9.405 s]
[INFO] Caliper API ........................................ SUCCESS [  4.635 s]
[INFO] Caliper Core ....................................... SUCCESS [  5.382 s]
[INFO] Caliper Runner ..................................... SUCCESS [ 10.345 s]
[INFO] Caliper Worker ..................................... SUCCESS [  4.077 s]
[INFO] Caliper Android Worker ............................. SUCCESS [  3.168 s]
[INFO] Caliper Android .................................... SUCCESS [  3.314 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 42.656 s
[INFO] Finished at: 2018-05-16T03:06:27+03:00
[INFO] Final Memory: 41M/154M
[INFO] ------------------------------------------------------------------------
```

Рисунок 3.25 — Фрагмент процесу зборки Google Caliper

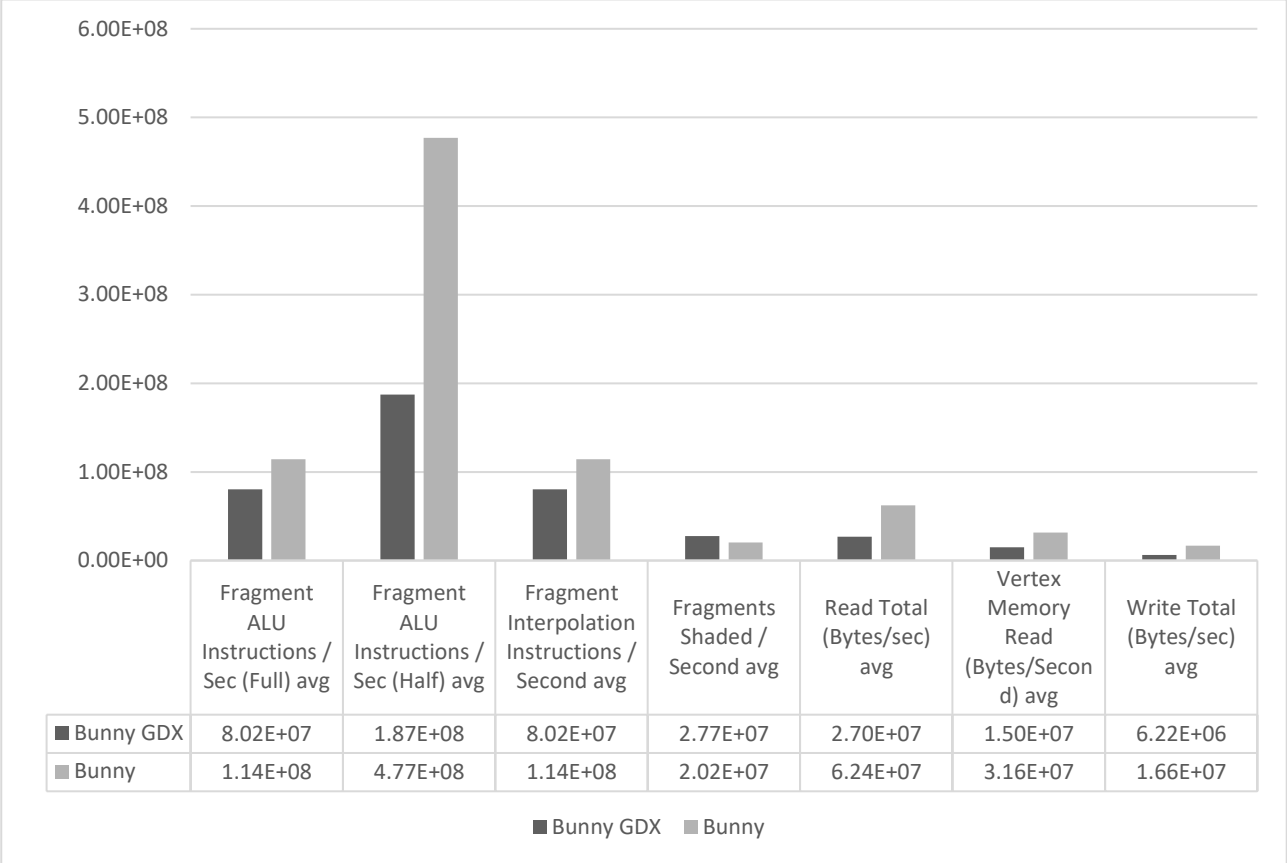
3.10 Порівняння з існуючими реалізаціями

В якості основної альтернативної існуючої реалізації в рамках даного дослідження стеку виступає фреймворк libGDX, розглянутий у підрозділі 1.4.1. libGDX, в силу використаних внутрішніх механізмів, пов’язаних з особливостями використання IBO, має серйозне обмеження на максимальну кількість вершин в межах однієї моделі, це обмеження складає 32767 — 65535 одиниць в залежності від реалізації [64]. В огляду на це, профілювання було проведення з використанням іншої стенфордської моделі — стенфордського кролика. Результати бенчмарку (рисунок 3.26, 3.27, 3.28, 3.29):



	% Shaders Busy avg	% Stalled on System Memory avg	% Vertex Fetch Stall avg	CPU Utilization % avg	FPS avg
Bunny GDX	3.76	0.14	0.35	7.74	54.51
Bunny	1.40	0.17	0.42	5.97	55.87

Рисунок 3.26 — Результати бенчмарку



	Fragment ALU Instructions / Sec (Full) avg	Fragment ALU Instructions / Sec (Half) avg	Fragment Interpolation Instructions / Second avg	Fragments Shaded / Second avg	Read Total (Bytes/sec) avg	Vertex Memory Read (Bytes/Second) avg	Write Total (Bytes/sec) avg
Bunny GDX	8.02E+07	1.87E+08	8.02E+07	2.77E+07	2.70E+07	1.50E+07	6.22E+06
Bunny	1.14E+08	4.77E+08	1.14E+08	2.02E+07	6.24E+07	3.16E+07	1.66E+07

Рисунок 3.27 — Результати бенчмарку

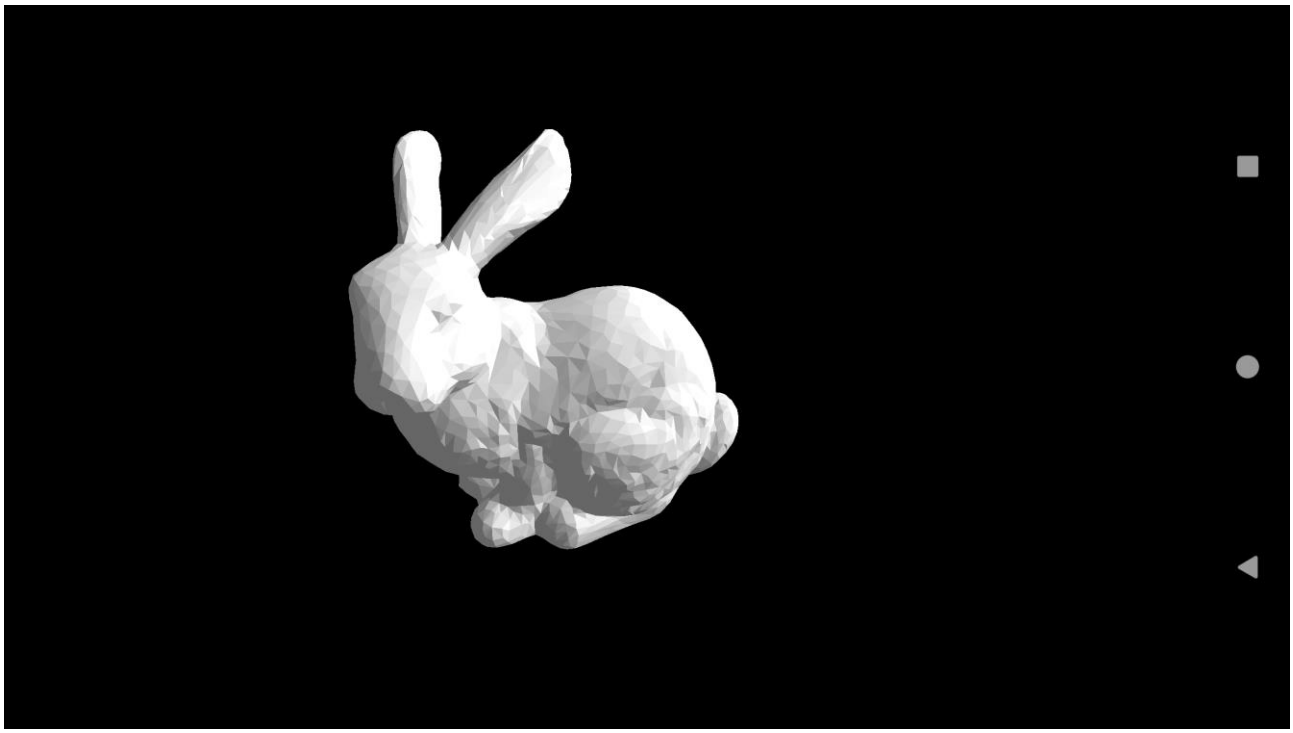


Рисунок 3.28 — Знімок екрану бенчмарку libGDX

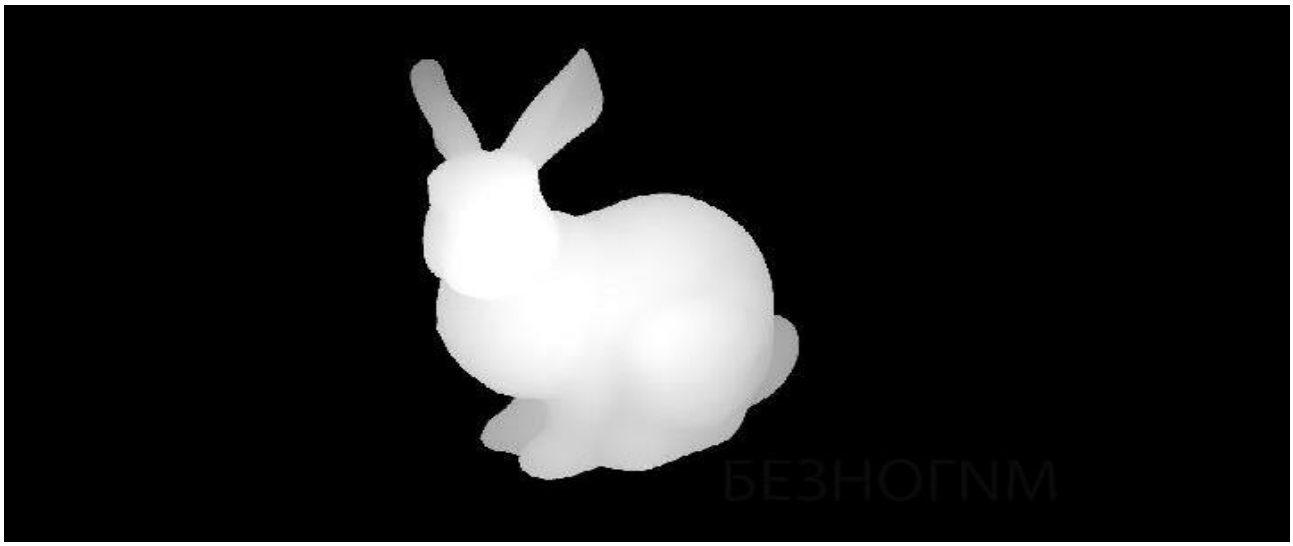


Рисунок 3.29 — Знімок екрану пропрієтарного бенчмарку

Як видно з результатів профілювання, пропрієтарне рішення показало кращі результати в контексті ефективності за тих же умов середовища та за даних особливостей рендерингу. Огляд метрик профілювання було наведено в підрозділі 3.3. В якості перспектив дослідження пропонується зробити огляд та оптимізацію існуючої системи в контекстні багатомодельного рендерингу, що знаходиться за рамками даного дослідження на даний момент.

РОЗДІЛ 4. ПРОЕКТ ЯК СТАРТАП

4.1 Опис ідеї проекту

Результати даного дослідження можна використати для створення рушія тривимірного рендерингу під управлінням операційної системи Android на комерційній основі.

Таблиця 4.1 — Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Рушій тривимірного рендерингу під управлінням	1. Ігрова розробка	1. Ефективне рішення для рендерингу ігрового додатку

операційної системи Android		2. Зменшення витрат часу на дослідження, розробку і тестування пропріетарних рішень
	2. Розробка рушіїв	1. Ефективне рішення для застосування в складі багатофункціонального рушія в аспекті тривимірного рендерингу 2. Зменшення витрат часу на дослідження, розробку і тестування пропріетарних рішень

Висновок: в таблиці приведені основні напрямки застосування рушія тривимірного рендерингу під управлінням операційної системи Android. Споживачами є розробники прикладного програмного забезпечення та комплексних рушіїв.

Таблиця 4.2 — Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів		W	N	S
		Мій проект	Комерційний конкурент			
1.	Ефективність	висока	середня			+
2.	Функціональність	середня	висока		+	
3.	Вартість	адаптивна	фіксована, висока			+
4.	Багатоплатформність	2 платформи	5 платформ	+		
5.	Легкість освоєння	Присутнє	Присутнє		+	
6.	Гнучкі тарифні плани	Присутні	Відсутні			+
7.	Надання початкового коду	Присутнє	Відсутнє			+
8.	Наявність служби технічної підтримки	Присутня	Присутня		+	

У порівнянні із головними конкурентами товар має ряд переваг — це ефективність, легкість освоєння та гнучкі тарифні плани.

4.2 Технологічний аудит ідеї проекту

Таблиця 4.3 — Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Багатоплатформність	Багатоплатформний фреймворк в основі рушія	Наявні	Доступні
		Залежні від платформи реалізації	Наявні	Доступні
2	Специфікація прикладного програмного інтерфейсу тривимірного рендерингу	OpenGL ES 1.0+	Наявні	Доступні
		OpenGL ES 2.0+	Наявні	Доступні
		OpenGL ES 3.0+	Наявні	Доступні
		Vulkan	Наявні	Доступні
3	Ефективні модулі рендерингу	Стек технологій, що об’єднує реалізації на рівні SDK та NDK	Необхідна розробка	Доступні
		Стек технологій на основі NDK	Необхідна розробка	Доступні
Обрана технологія реалізації ідеї проекту: залежні від платформи реалізації рушія на основі прикладного програмного інтерфейсу тривимірного рендерингу OpenGL ES 3.0 зі стеком технологій, що об’єднує реалізації на рівні SDK та NDK				

За результатами аналізу таблиці робимо висновок щодо можливості технологічної реалізації проекту.

4.3 Аналіз ринкових можливостей запуску стартап-проекту

Таблиця 4.4 — Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	2
2	Загальний обсяг продаж, ум.од	100000
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Відсутні
5	Специфічні вимоги до стандартизації та сертифікації	Відсутні
6	Середня норма рентабельності в галузі (або по ринку), %	72%

За результатами аналізу ринкових можливостей запуску стартап-проекту можна зробити висновок про привабливість ринку для входження за попереднім оцінюванням. Мала кількість гравців свідчить про високий поріг входу та малу конкуренцію при виборі правильного вектору розвитку.

Таблиця 4.5 — Характеристика потенційних клієнтів стартап-проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Ефективний тривимірний рендеринг	1. Розробники ігрових продуктів 2. Розробники ігрових рушіїв 3. Розробники прикладного програмного забезпечення	Використання у складі власного продукту для кінцевих користувачів чи у складі однотипного продукту з ширшим функціоналом	1. Високі показники ефективності рендерингу 2. Легкість інтеграції 3. Гнучкість реалізації 4. Гнучкі тарифні плани

Таблиця 4.6 — Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Конкуренція	Менша ціна у конкурента при	Відтік клієнтів

		однаковій якості продукту	
2	Якість	Низька якість інтеграції	Відмова від продукту
3	Функціонал	Недостача наявного функціоналу рушія для задоволення потреб	Відмова від продукту

Висновки: головними факторами загроз є конкуренція та якість інтеграції. Вже існуючі товари на ринку мають певну репутацію та контракти на постачання оновлень і підтримку у споживачів. Конкуренти здатні демпінгувати ціни для отримання нових клієнтів свого товару. Нові клієнти власного продукту можуть здійснити інтеграцію рушія з низьким рівнем якості в силу внутрішніх факторів компанії-замовника.

Таблиця 4.7 — Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Отримання необхідних інвестицій	Сформований початковий капітал, необхідний для реалізації мінімально життєздатного продукту	Розробка мінімально життєздатного продукту
2	Освоєння нових сфер	Використання підсистеми у нових, досі не розглянутих сферах	Створення спеціальної робочої групи задля модернізації підсистеми для виконання нових вимог
3	Співпраця з відомими розробниками комплексних рушіїв	Розглядається співпраця з метою використання розробленої підсистеми у складі комплексних рушіїв	Масштабування функціоналу підсистеми
4	Успішна маркетингова політика	В результаті проведеної маркетингової	Підтримка стабільної роботи системи та її розвитку

		політики отримана висока зацікавленість користувачів	Збільшення цін на використання рушія
5	Ліквідація конкурента	Конкурент ліквідував свою компанію у результаті власного бажання або зовнішніх чинників	Проведення маркетингової кампанії для монополізації ринку
6	Індивідуальне замовлення	Клієнт потребує надбудови нового специфічного функціоналу	Оцінка затрат і вигоди компанії в даній ситуації. Погодження умов можливого контракту.

Висновки: сфера використання таких рушіїв швидко розвивається, ринок клієнтів постійно зростає. Збільшення зацікавленості в товарі призведе до різкого збільшення об’ємів постачання та продажів, що дасть поштовх до розробки нового функціоналу. Це досягається шляхом рекламування та освоєння нових сфер використання рушіїв.

Таблиця 4.8 — Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Тип конкуренції: Олігополія	Незначна кількість конкурентів Велика ринкова сила Схожість використовуваних технологій	Інформування ринку щодо появи нової системи Співпраця із провідними конкурентами
2. За рівнем конкурентної боротьби: Галузева	Загроза появи нових конкурентів Ринкова влада споживачів Висока потреба у товарі	Інформування ринку щодо якості використовуваної новаторської технології Пропозиція гнучких цін

3. За галузевою ознакою: Внутрішньогалузева	Діяльність в одній галузі економіки Надання продуктів одного типу	Зменшення вартості сервісу Примноження каналів розподілу
4. Конкуренція за видами товарів: Товарно-видова	Надання різних продуктів одного виду	Маркетингова політика
5. За характером конкурентних переваг: Цінова	Використання цін для покращення економічних умов збуту	Зменшення вартості сервісу Використання нових каналів розподілу
6. За інтенсивністю: Марочна	Пропозиція схожого сервісу Спільна цільова аудиторія	Інформування ринку щодо якості використовуваної новаторської технології Примноження каналів розподілу

Таблиця 4.9 — Аналіз конкуренції в галузі за М. Портером

	Прямі конкуренти в галузі	Потенційні конкуренти	Клієнти	Товари-замінники
Складові аналізу	«Unity», «Unreal»	Високі бар’єри входження на ринок	Рівень чутливості до цін, продуктова диференціація: якість, спосіб отримання продукту, швидкість обслуговування	Копіювання функціоналу, Мінімізація цін
Висновки:	CR4 = 75% Індекс Херфіндаля-Хіршмана (HHI) = 6236 Значення показників вказує на високу концентрацію (монополізаці	Можливості виходу на ринок малі, потенційні конкуренти на даний момент відсутні	Клієнти диктують умови гнучкості цінової політики, високої і довгострокової якості підтримки продукту та наявність кооперації із сервісами та продуктами	Пропонування вигідних умов, забезпечення захисту інтелектуальної власності, гнучкість цінової політики

	ю) даного ринку		суміжних компаній, що вони використовують	

Висновки: за результатами аналізу конкуренції можна зробити висновок щодо принципової можливості роботи на ринку з огляду на конкурентну ситуацію.

Таблиця 4.10 — Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Унікальність системи	Система забезпечує високу ефективність тривимірного рендерингу
2	Модель бізнес для бізнесу	Бізнес модель ґрунтується на співпраці із сервісами, що надають багатофункціональні рушії
3	Цінова політика	Отримання прибутку здійснюється за рахунок користувачів або отримання процентів з прибутку сторонніх сервісів. Даний підхід дозволить обійти цінову конкуренцію на ринку цільової аудиторії

Висновки: оцінено основні фактори конкурентної спроможності. Система забезпечує високу ефективність тривимірного рендерингу завдяки інноваційним технікам та особливостям. Простота інтеграції, гнучкість реалізації та цінова політика робить продукт більш привабливим для клієнта.

Таблиця 4.11 — Порівняльний аналіз сильних та слабких сторін

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з конкурентом						
			–3	–2	–1	0	+1	+2	+3
1	Унікальність сервісу	20							+

2	Модель бізнес для бізнесу	15						+	
3	Цінова політика	10					+		

Висновки: спираючись на фактори конкурентоспроможності та підсумовуючи рейтинг товару відносно головного конкурента, запропонований продукт має більший рейтинг відносно прямих конкурентів. Дана таблиця показує якими саме особливостями розроблений продукт відрізняються від аналогів та в яку саме сторону.

Таблиця 4.12 — SWOT-аналіз стартап-проекту

Сильні сторони: Якість продукту Низькі ціни Додаткові можливості інтеграції	Слабкі сторони: Недостача стартових капіталовкладень Бізнес модель залежить від політики окремих бізнесів Необхідність стрімкого входу на ринок
Можливості: Інвестиції Реалізація бізнес-моделей Висока зацікавленість цільової аудиторії	Загрози: Крадіжка інтелектуальної власності Відмова суміжних компаній у співпраці

Таблиця 4.13 — Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Створення власного багатофункціонального рушія	Мало ймовірне	6 місяців
2	Маркетингова кампанія для приваблювання користувачів	Ймовірне	2 місяці

3	Пропонування безкоштовних версій продукту	Ймовірне	4 місяці
4	Пошук бізнесів іншої галузі для співпраці	Дуже ймовірне	4 місяців
Обрана альтернатива: Пошук бізнесів іншої галузі для співпраці			

4.4 Розроблення ринкової стратегії проекту

Таблиця 4.14 — Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачі в сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Незалежні розробники прикладних продуктів	Висока	Високий	Низька	Низькі бар’єри входу
2	Постачальники комплексних рішень-рушіїв	Висока	Середній	Низька	Високі бар’єри входу
Які цільові групи обрано: Незалежні розробники прикладних продуктів					

Таблиця 4.15 — Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку

1	Надання рушія розробникам прикладних продуктів	Вибірковий розподіл	Здатність протистояти прямим конкурентам Низькі витрати Ефективна співпраця посередників	Стратегія диференціації

Таблиця 4.16 — Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки*
1	Ні	Забирати та залучати нових	Надання підсистем рушія для сторонніх розробників	Стратегія лідера. Розширення первинного попиту

Таблиця 4.17 — Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)

1	Високі показники ефективності рендерингу Легкість інтеграції Гнучкість реалізації Гнучкі тарифні плани	Стратегія диференціації	Формування регулярного попиту Виявлення нових груп споживачів Нові напрями застосування існуючого продукту	Захист авторського права Інноваційність технології Простота використання

4.5 Розроблення маркетингової програми стартап-проекту

Таблиця 4.18 — Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1	Високі показники ефективності	Підвищення показників продуктивності тривимірного рендерингу	Якість послуги Інноваційність підходу Цінова перевага
2	Легкість інтеграції	Зменшення витрат часу на інтеграцію рішення	Інноваційність підходу Простота реалізації

Таблиця 4.19 — Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові
І. Товар за задумом	Рушій тривимірного рендерингу під управлінням операційної системи Android

II. Товар у реальному виконанні	Властивості/характеристики
	Комплексні метрики ефективності
	Якість: стандарти, рівень оптимізації написаного коду, коректність використаних технологій, звіти з тестування програмного забезпечення
	User Manual
	Марка: Engine
III. Товар із підкріпленням	До продажу
	Після продажу
За рахунок технік обфускації програмного коду товар буде захищеним від копіювання	

Висновки: основними засобами захисту від копіювання є використання технік обфускації програмного рішення. Закладені характеристики на другому та третьому рівнях товару робить його унікальним серед конкурентів.

Таблиця 4.20 — Визначення меж встановлення ціни

№ п/п	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	$625	$125	$20000	$100/$2500

Таблиця 4.21 — Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Закупівля здійснюється через довірені джерела	Інформування користувачів Доступ користування сервісом	Канал одного рівня	Селективна з використанням комбінованого

				каналу збуту

Таблиця 4.22 — Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Ведення бізнесу в сфері розробки і розповсюдження цифрового контенту Робота з цифровою продукцією	Прямі неофіційні	Послідовність в реалізації обраної позиції Доступність та об’єктивність інформації про фірму і товар Унікальність послуги	Формування у цільової аудиторії обізнаності про появу нового продукту Інформування користувачів про властивості та переваги продукту Інформування користувачів про нові способи використання відомого продукту	Раціоналістична стратегія реклами

4.6 Висновки до розділу

Даний розділ присвячений розробленню першого етапу створення стартап-проекту. Найголовнішим в проведенні будь-якої наукової роботи є подальша комерціалізація отриманих результатів та можливість застосування розробленої концепції в промисловості.

В результаті детального аналізу було виявлено, що пропонована ідея стартап-проекту має можливість ринкової комерціалізації з перспективами

впровадження з огляду на потенційні групи клієнтів. Розглянутий ринок має високий бар’єр входження та низьку конкуренцію, що, за наявності якісного, конкурентоспроможного продукту з вагомими сильними сторонами, унікальністю та незамінністю пропонованих особливостей і функцій, дає можливість швидко отримати ресурси, прибутки та довіру користувачів і стати одним із лідерів галузі з можливістю подальшого розвитку продукту.

ВИСНОВКИ

В рамках даного дослідження був обраний, детально проаналізований та частково імплементований стек ефективного тривимірного рендерингу під управлінням операційної системи Android на базі інтерфейсу OpenGL ES 3.0. В ході імплементації, конфігурації та профілювання технік та особливостей програмованих частин стеку, його вузьких місць, був отриманий ефективний в контексті поставленої задачі набір останніх, їх конкретні реалізації, налаштування, принципи взаємодії. Деякі розглянуті і запропоновані техніки, особливості та підходи дали негативні результати в контексті ефективності та можливості їх застосування та були заміщенні або опущені.

В результаті профілювання рішення наряду з існуючими аналогами за тих же умов були визначені позитивні показники ефективності.

Результати дослідження мають діюче комерційне застосування та є актуальними і перспективними в області систем автоматизованого проектування і розрахунку, віртуальної реальності, наукової візуалізації та відеоігор.

ПЕРЕЛІК ПОСИЛАНЬ

1. OpenGL - The Industry Standard for High Performance Graphics [Електронний ресурс] – Режим доступу до ресурсу: https://www.opengl.org/ (дата звернення: 20.05.2018) – Назва з екрана.

2. OpenGL ES - The Standard for Embedded Accelerated 3D Graphics [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengles/ (дата звернення: 20.05.2018) – Назва з екрана.

3. Android Developers [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/index.html (дата звернення: 20.05.2018) – Назва з екрана.

4. 3D RENDERING SOFTWARE [Електронний ресурс] – Режим доступу до ресурсу: https://www.autodesk.com/solutions/3d-rendering-software (дата звернення: 20.05.2018) – Назва з екрана.

5. Graphics Processing Unit (GPU) [Електронний ресурс] – Режим доступу до ресурсу: http://www.nvidia.com/object/gpu.html (дата звернення: 20.05.2018) – Назва з екрана.

6. Hardware acceleration [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/guide/topics/graphics/hardware-accel (дата звернення: 20.05.2018) – Назва з екрана.

7. What’s the Difference Between a CPU and a GPU? [Електронний ресурс] – Режим доступу до ресурсу: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (дата звернення: 20.05.2018) – Назва з екрана.

8. Android [Електронний ресурс] – Режим доступу до ресурсу: https://www.android.com/ (дата звернення: 20.05.2018) – Назва з екрана.

9. Android Captures Record 85 Share of Global Smartphone Shipments in Q2 2014 [Електронний ресурс] – Режим доступу до ресурсу: http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=9921 (дата звернення: 20.05.2018) – Назва з екрана.

10. Distribution dashboard [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/about/dashboards/ (дата звернення: 20.05.2018) – Назва з екрана.

11. Learning about the Android graphics subsystem - Imagination Blog [Електронний ресурс] – Режим доступу до ресурсу: http://blog.imgtec.com/news/learning-about-the-android-graphics-subsystem (дата звернення: 20.05.2018) – Назва з екрана.

12. Support different screen sizes [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/training/multiscreen/screensizes (дата звернення: 20.05.2018) – Назва з екрана.

13. What is the “Texture Fill-Rate” on a GPU and Does it Matter? [Електронний ресурс] – Режим доступу до ресурсу: https://www.gamersnexus.net/guides/1747-what-is-texture-fill-rate-defined (дата звернення: 20.05.2018) – Назва з екрана.

14. Skia Graphics Library [Електронний ресурс] – Режим доступу до ресурсу: https://skia.org/ (дата звернення: 20.05.2018) – Назва з екрана.

15. Android HWUI硬件加速模块浅析 [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/DsoTsin/AndroidDev/blob/master/Android%20HWUI%E7%A1%AC%E4%BB%B6%E5%8A%A0%E9%80%9F%E6%A8%A1%E5%9D%97%E6%B5%85%E6%9E%90.md (дата звернення: 20.05.2018) – Назва з екрана.

16. RenderScript Overview [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/guide/topics/renderscript/compute (дата звернення: 20.05.2018) – Назва з екрана.

17. Surface [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/reference/android/view/Surface (дата звернення: 20.05.2018) – Назва з екрана.

18. SurfaceFlinger and Hardware Composer [Електронний ресурс] – Режим доступу до ресурсу: https://source.android.com/devices/graphics/arch-sf-hwc (дата звернення: 20.05.2018) – Назва з екрана.

19. OpenGL Overview [Електронний ресурс] – Режим доступу до ресурсу: https://www.opengl.org/about/ (дата звернення: 20.05.2018) – Назва з екрана.

20. OpenGL® ES 3.0 Reference Pages [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/registry/OpenGL-Refpages/es3.0/ (дата звернення: 20.05.2018) – Назва з екрана.

21. OpenGL ES [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/guide/topics/graphics/opengl (дата звернення: 20.05.2018) – Назва з екрана.

22. Khronos Vulkan Registry [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/registry/vulkan/ (дата звернення: 20.05.2018) – Назва з екрана.

23. Desktop/Android/BlackBerry/iOS/HTML5 Java game development framework [Електронний ресурс] – Режим доступу до ресурсу: https://libgdx.badlogicgames.com/ (дата звернення: 20.05.2018) – Назва з екрана.

24. Cortex-A7 [Електронний ресурс] – Режим доступу до ресурсу: https://developer.arm.com/products/processors/cortex-a/cortex-a7 (дата звернення: 20.05.2018) – Назва з екрана.

25. Android NDK [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/ndk/ (дата звернення: 20.05.2018) – Назва з екрана.

26. Rendering Pipeline Overview [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (дата звернення: 20.05.2018) – Назва з екрана.

27. Vertex Specification [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Vertex_Specification (дата звернення: 20.05.2018) – Назва з екрана.

28. Primitive [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Primitive (дата звернення: 20.05.2018) – Назва з екрана.

29. OpenGL Object [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/OpenGL_Object (дата звернення: 20.05.2018) – Назва з екрана.

30. GLSL Object [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/GLSL_Object (дата звернення: 20.05.2018) – Назва з екрана.

31. Shader [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Shader (дата звернення: 20.05.2018) – Назва з екрана.

32. Vertex Shader [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Vertex_Shader (дата звернення: 20.05.2018) – Назва з екрана.

33. Fragment Shader [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Fragment_Shader (дата звернення: 20.05.2018) – Назва з екрана.

34. Shader Compilation [Електронний ресурс] – Режим доступу до ресурсу: https://www.opengl.org/wiki/Shader_Compilation (дата звернення: 20.05.2018) – Назва з екрана.

35. Uniform (GLSL) [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Uniform_(GLSL) (дата звернення: 20.05.2018) – Назва з екрана.

36. Depth Test [Електронний ресурс] – Режим доступу до ресурсу: https://www.opengl.org/wiki/Depth_Test (дата звернення: 20.05.2018) – Назва з екрана.

37. OpenGL Type [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Type_Qualifier_(GLSL) (дата звернення: 20.05.2018) – Назва з екрана.

38. Affine Transformation [Електронний ресурс] – Режим доступу до ресурсу: http://mathworld.wolfram.com/AffineTransformation.html (дата звернення: 20.05.2018) – Назва з екрана.

39. Marcel B. Geometry I / Berger Marcel. – Berlin: Springer, 1987.

40. Nomizu K. Affine Differential Geometry (New ed.) / K. Nomizu, S. S., 1994.

41. Shreiner D. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V (9th Edition) / Dave Shreiner., 2016.

42. Buffer Object Streaming [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Buffer_Object_Streaming (дата звернення: 20.05.2018) – Назва з екрана.

43. Java Native Interface Specification [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html (дата звернення: 20.05.2018) – Назва з екрана.

44. The Stanford 3D Scanning Repository [Електронний ресурс] – Режим доступу до ресурсу: http://graphics.stanford.edu/data/3Dscanrep/ (дата звернення: 20.05.2018) – Назва з екрана.

45. blender.org - Home of the Blender project - Free and Open 3D Creation Software [Електронний ресурс] – Режим доступу до ресурсу: https://www.blender.org/ (дата звернення: 20.05.2018) – Назва з екрана.

46. Graphics API Debugger [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/google/gapid (дата звернення: 20.05.2018) – Назва з екрана.

47. Snapdragon Profiler [Електронний ресурс] – Режим доступу до ресурсу: https://developer.qualcomm.com/software/snapdragon-profiler (дата звернення: 20.05.2018) – Назва з екрана.

48. Google Pixel XL [Електронний ресурс] – Режим доступу до ресурсу: https://www.gsmarena.com/google_pixel_xl-8345.php (дата звернення: 20.05.2018) – Назва з екрана.

49. Vertex Array Object [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opengl/wiki/Vertex_Specification#Vertex_Array_Object (дата звернення: 20.05.2018) – Назва з екрана.

50. Class Buffer [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/7/docs/api/java/nio/Buffer.html (дата звернення: 20.05.2018) – Назва з екрана.

51. Buffer [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/reference/java/nio/Buffer (дата звернення: 20.05.2018) – Назва з екрана.

52. JNI Functions [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html (дата звернення: 20.05.2018) – Назва з екрана.

53. Class System [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/7/docs/api/java/lang/System.html (дата звернення: 20.05.2018) – Назва з екрана.

54. The Java® Language Specification [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/specs/jls/se10/html/index.html (дата звернення: 20.05.2018) – Назва з екрана.

55. Don’t Get Caught In the Cold, Warm-up Your JVM [Електронный ресурс] / [D. Lion, A. Chiu, H. Sun та ін.]. – 2016. – Режим досту-пу до ресурсу: http://www.eecg.toronto.edu/~yuan/papers/osdi16-hottub.pdf (дата звернення: 20.05.2018).

56. Snapdragon 821 Mobile Platform [Електронний ресурс] – Режим доступу до ресурсу: https://www.qualcomm.com/products/snapdragon/processors/821 (дата звернення: 20.05.2018) – Назва з екрана.

57. ProGuard [Електронний ресурс] – Режим доступу до ресурсу: https://www.guardsquare.com/en/proguard (дата звернення: 20.05.2018) – Назва з екрана.

58. oreo-release - platform/frameworks/base.git - Git at Google [Електронний ресурс] – Режим доступу до ресурсу: https://android.googlesource.com/platform/frameworks/base.git/+/oreo-release (дата звернення: 20.05.2018) – Назва з екрана.

59. Micro-benchmarking library for Java [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/google/caliper (дата звернення: 20.05.2018) – Назва з екрана.

60. Kurzyniec D. Efficient Cooperation between Java and Native Codes – JNI Performance Benchmark [Електронний ресурс] / D. Kurzyniec, V. Sunderam – Режим доступу до ресурсу: https://pdfs.semanticscholar.org/2b7e/9b075e51c5eb51bb035b39b17617f7428247.pdf

61. Android Studio features | Android Developers [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/studio/features/ (дата звернення: 20.05.2018) – Назва з екрана.

62. FloatBuffer [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/reference/java/nio/FloatBuffer (дата звернення: 20.05.2018) – Назва з екрана.

63. The leading operating system for PCs, IoT devices, servers and the cloud | Ubuntu [Електронний ресурс] – Режим доступу до ресурсу: https://www.ubuntu.com/ (дата звернення: 20.05.2018) – Назва з екрана.

64. Importing Blender models in LibGDX [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/libgdx/libgdx/wiki/Importing-Blender-models-in-LibGDX#maximum-vertices (дата звернення: 20.05.2018) – Назва з екрана.

ДОДАТОК А

НАЦІОНАЛЬНА АКАДЕМІЯ НАУК УКРАЇНИ
ІНСТИТУТ ПРОГРАМНИХ СИСТЕМ

ISSN 1727-4907

ПРОБЛЕМИ ПРОГРАМУВАННЯ

НАУКОВИЙ ЖУРНАЛ

PROBLEMS IN PROGRAMMING

SCIENTIFIC JOURNAL

2018

№ 1

Теми випуску:

- *Теоретичні та методологічні основи програмування*
- *Моделі та засоби паралельних і розподілених програм*
- *Інструментальні засоби та середовища програмування*
- *Методи та засоби програмної інженерії*
- *Формальні методи розробки програмного забезпечення*
- *Моделі та засоби систем баз даних і знань*
- *Експертні та інтелектуальні інформаційні системи*
- *Прикладні засоби програмування та програмне забезпечення*
- *Математичне моделювання об'єктів та процесів*
- *Методи, моделі та стандарти створення структурованих електронних документів*
- *Наукова інформація*

НАЦІОНАЛЬНА АКАДЕМІЯ НАУК УКРАЇНИ
ІНСТИТУТ ПРОГРАМНИХ СИСТЕМ

ПРОБЛЕМИ ПРОГРАМУВАННЯ

науковий журнал

Головний редактор
Андон Пилип Іларіонович
академік НАН України,
директор Інституту програмних систем
НАН України
✉ Інститут програмних систем
НАН України
проспект Академіка Глушкова, 40, корп. 5
03187, Київ-187
☎ Тел.+380 (44) 526 5507
E-mail: andon@isofts.kiev.ua
http://www.pp.isofts.kiev.ua

Адреса для кореспонденції

✉ Інститут програмних систем
НАН України
Проспект Академіка Глушкова, 40
03187, Київ-187

☎ Тел.: +380 (44) 526 5065
Факс: +380 (44) 526 6263
E-mail: iss@isofts.kiev.ua

Редактор *В.П. Замула*
Комп'ютерна верстка *В.П. Замула*

Підписано до друку 28.02.2018. Формат 60x84/8. Ум. друк. арк. 18,60.
Обл.-вид. арк. 16,06. Тираж 120 прим. Ціна договірна. Замовл. 5181

Віддруковано ВД «Академперіодика» НАН України
вул. Терещенківська, 4, м. Київ, 01004

Свідоцтво суб'єкта видавничої справи ДК № 544 від 27.07.2001

НАЦІОНАЛЬНА АКАДЕМІЯ НАУК УКРАЇНИ
ІНСТИТУТ ПРОГРАМНИХ СИСТЕМ

ПРОБЛЕМИ ПРОГРАМУВАННЯ

науковий журнал

№ 1 січень-березень 2018

Заснований у березні 1999 р.

ЗМІСТ

УДК 681.3

І.З. Ашур, А.Ю. Дорошенко

ВИСОКОПРОДУКТИВНИЙ ПАКЕТНИЙ ДОБУТОК МАТРИЦЬ АФІННИХ ПЕРЕТВОРЕНЬ ЗА ДОПОМОГОЮ ANDROID NDK ТА JNI

Робота містить опис високопродуктивного підходу для здійснення пакетного добутку матриць афінних перетворень за допомогою Android NDK та JNI. Розібрані основні використані техніки та особливості, проведена оцінка продуктивності підходу у порівнянні з його альтернативами і попередниками.

Ключові слова: Android NDK, Android SDK, JDK, JVM, JNI, OpenGL ES, Java, афінні перетворення.

Вступ

Ефективний добуток матриць афінних перетворень – одна з актуальних проблем розробки додатків для мобільних платформ, що використовують високопродуктивну двовимірну комп'ютерну графіку.

Такі додатки розробляються для ігрової індустрії, програмного забезпечення візуалізації, моделювання в частині рендерингу візуальних моделей, візуальної інтерактивної взаємодії.

Одним з аспектів використання результатів даних обчислень є добуток матриць виду, моделі і проекції.

Матричне представлення використовується, зокрема, для запису афінних перетворень в комп'ютерній графіці.

Мета даної роботи – огляд результатів, отриманих за прогресом розробки високопродуктивних рішень для пакетного добутку матриць афінних перетворень у просторі Android SDK, NDK, JDK, JNI.

1. Аналіз предметної області

Афінні перетворення зазвичай використовуються у лінійній алгебрі для представлення лінійних перетворень, а векторна сума – для представлення паралельних перенесень. Використовуючи розширену матрицю та розширений вектор стає можливим представлення лінійного переносу та лінійних перетворень з використанням одного добутку матриць. Ця техніка потребує розширення всіх векторів додатковою «1» в кінці, а всіх матриць – додатковим рядком нулів знизу та додатковим стовпчиком – вектором переносу – справа, а також, «1» в правому нижньому кутку.

Звичайний матрично-векторний добуток завжди відображає початок координат на початок координат, а тому ніколи не може представляти лінійного переносу, в якому початок координат має бути відображеним до якоїсь іншої точки. Завдяки додаванню додаткової координати зі значенням «1» до кожного вектора, останній представляє відображення простору як підмножини простору з додатковим виміром. В цьому просторі початковий простір займає підмножину, де додаткова координата дорівнює 1. Виходячи з цього, початок координат початкового простору знаходиться в (0,0,…,0,1). Таким чином стає можливим лінійний перенос початкового простору засобами лінійного перетворення простору з більшим числом вимірів. Координати в просторі з більшим числом вимірів є прикладом однорідних координат [1, 2].

Завдяки використанню системи однорідних координат стає можливим застосування комбінації будь-якої кількості афінних перетворень одним добутком відповідних матриць. Ця властивість широко використовується в комп'ютерній графіці, засобах комп'ютерного зору та робототехніці.

Матриця виду описує представлення афінного перетворення як-то: стиснення, розтягнення, поворот, паралельний пе-

ренос, відображення та інші окремі та загальні випадки трансформацій.

Простір виду представляє результат перетворення світових координат у координати видимого простору. Видимий простір у даному випадку визначається сукупністю перетворень здвигів і поворотів сцени. Ці комбіновані перетворення представляються матрицею виду.

Сучасні системи рендерингу двовимірної графіки також оперують поняттям простору відсічення. Простір відсічення визначає видимі області сцени і, як наслідок, вершини [3].

Загальний процес використання перетворень показаний на рис. 1:

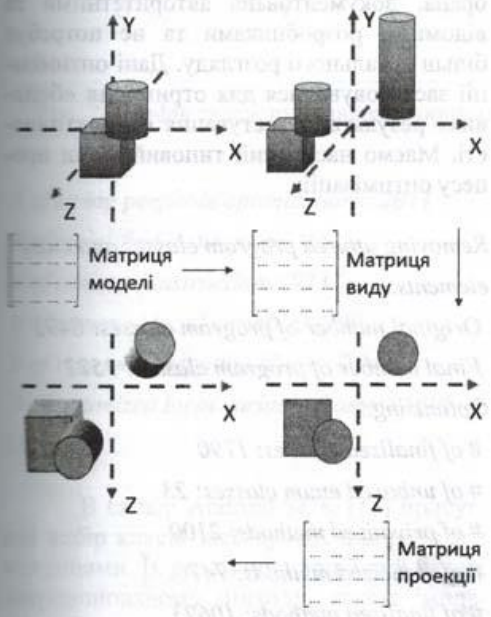


Рис. 1. Послідовність використання матричних перетворень

Деталі застосування даних трансформацій та їх математичний зміст не є предметом обговорення даної статті і приведені з ціллю надання загальної інформації про предметну область вирішуваної задачі.

Дане дослідження націлене на розробку, створення і аналіз продуктивності підходу добутку матриць афінних перетворень за допомогою Android NDK [4] та JNI [5].

2. Аналіз та розробка рішень

Відштовхуючись від введених обмежень предметної області у виді матриць афінних перетворень, надалі будемо оперувати матрицями розміру 4×4 [6].

Класична, наївна реалізація добутку матриць передбачає використання стандартних засобів JDK [7] з використанням трьох вкладених циклів.

```
int a[][] = {{}, {}, {}};
int b[][] = {{}, {}, {}, {}};
int al = a.length;
int bl = b[0].length;
int [][] result = new int[al][bl];

for (int i = 0; i < al; i++) {
  for (int j = 0; j < bl; j++) {
    for (int k = 0; k < al; k++) {
      result[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

Для оцінки продуктивності підходу та його наївних аналогів були створені спеціальні програми-бенчмарки з урахуванням всіх «best practices». Існує багато оптимізації, що можуть бути використані віртуальною Java-машиною або апаратним забезпеченням до конкретного компонента під час його ізольованого тестування. Непродумані тести продуктивності можуть давати оптимістичні результати, що зовсім відрізняються від реальних. Наприклад, навіть порядок виконання тестів та кількість ітерацій тестування можуть створювати велику похибку в користь певного з підходів, шо тестуються.

Саме тестування конкретного компонента в складі робочого, практичного додатку з використанням таких технік, як JVM warm up (підігрів віртуальної Java-машини) [8] може дати достовірні результати.

Далі будуть приведені результати тестів продуктивності для ітерацій розроб-

леного підходу та його наївних альтернатив з різним стеком використаних технік та особливостей, їх порівняння. Під метрикою продуктивності мається на увазі час, затрачений процесором на виконання певної кількості матричних добутків.

Результати тестування затрат процесорного часу на добуток 1 мільйона матриць 4×4 у виді одновимірного масиву на 16 елементів типу з рухомою комою за використанням класичної, наївної реалізації засобами JDK на пристрої Google Pixel XL [9] в 10 ітерацій представлені в табл. 1. та рис 2.

Табл. 1. Результати тестування продуктивності наївного підходу

Ітерація, №	Затрати процесорного часу, мс
1	3121
2	3141
3	3132
4	3155
5	3137
6	3127
7	3133
8	3115
9	3111
10	3116
Усереднений результат	3129

Результати тестів для різної кількості матриць, різної кількості ітерацій, різних пристроїв та різної розмірності матриць опускаються за причини практично прямої залежності між приведеними метриками та продуктивністю обчислень за умови відсутності вузьких місць конкретної платформи для тестування.

Представлені результати отримані на пристрої з процесором Qualcomm MSM8996 Snapdragon 821 Quad-core (2x2.15 GHz Kryo & 2x1.6 GHz Kryo) [10].

Слід зазначити, що тести продуктивності проводились на збірках додатку з використанням агресивної оптимізації та обфускації сирців за використанням інструменту ProGuard [11] з сімома проходами оптимізації, що, зокрема, застосовують арифметичні оптимізації, фіналізацію класів, їх членів, вертикальне та горизонтальне склеювання класів, приватизацію полів, методів, витягнення статичних методів, оптимізацію кількості параметрів методів, вбудовування константних виразів, коротких та унікальних методів, набір методів «peephole optimization» [12] для арифметичних операцій, гілок коду, строкових даних, приведення типів, чистку надлишкових інструкцій, виключень, блоків коду.

Тема оптимізації продуктивності Java-коду на етапі компіляції достатньо вивчена на даний момент та глибоко розібрана, документована авторитетними та відомими розробниками та не потребує більш детального розгляду. Дані оптимізації застосовувалися для отримання «бойових» результатів тестування продуктивності. Маємо наступний типовий вивід процесу оптимізації:

Removing unused program classes and class elements...
Original number of program classes: 8493
Final number of program classes: 1522
Optimizing...
# of finalized classes: 1790
# of unboxed enum classes: 23
# of privatized methods: 2100
# of staticized methods: 747
# of finalized methods: 10693
# of removed method parameters: 713
# of inlined constant parameters: 277
# of inlined constant return values: 103
# of inlined short method calls: 3242
# of inlined unique method calls: 4711
# of inlined tail recursion calls: 43
# of merged code blocks: 155
# of variable peephole optimizations: 15645
# of field peephole optimizations: 166
# of branch peephole optimizations: 5619

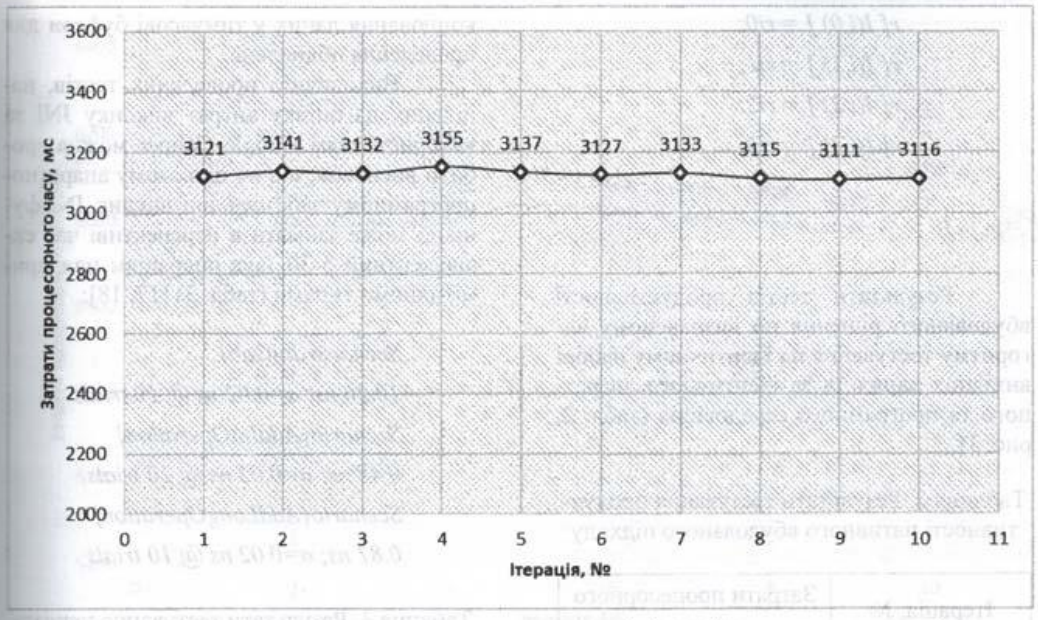


Рис. 2. Графік залежності затрат процесорного часу від ітерації тестування для наївного підходу

```
# of string peephole optimizations: 2011
# of simplified instructions: 1521
# of removed instructions: 9233
# of removed local variables: 599
# of removed exception blocks: 264
# of optimized local variable frames: 5395
Shrinking...
```

В складі Android SDK [13] присутній набір класів-застосувань для роботи з матрицями. Їх реалізація, подібно й надалі запропонованому підходу, також імплементована на рівні NDK. В цілому, вона повністю дублює зазначений вище підхід за виключенням наявності цілого ряду додаткового захисту і перевірок, інструкцій для очистки пам'яті.

Java Native Interface (JNI) є стандартним механізмом запуску C/C++ коду під керуванням віртуальної машини Java (JVM) [14].

Android NDK (Android Native Development Kit) – набір інструментів для розробки компонентів Android, базований на C/C++.

Як приклад буде приведений фрагмент зазначеної реалізації з ядра Android: /base/core/jni/android/opengl/util.cpp за станом на версію Oreo 8.0.0_r4 [15]:

```
#define I(_i, _j) ((_j)+ 4*(_i))

static
void multiplyMM(float* r,
                const float* lhs,
                const float* rhs) {
  for (int i=0 ; i<4 ; i++) {
    const float rhs_i0 = rhs[ I(i,0) ];
    float ri0 = lhs[ I(0,0) ] * rhs_i0;
    float ri1 = lhs[ I(0,1) ] * rhs_i0;
    float ri2 = lhs[ I(0,2) ] * rhs_i0;
    float ri3 = lhs[ I(0,3) ] * rhs_i0;
    for (int j=1 ; j<4 ; j++) {
      const float rhs_ij = rhs[ I(i,j) ];
      ri0 += lhs[ I(j,0) ] * rhs_ij;
      ri1 += lhs[ I(j,1) ] * rhs_ij;
      ri2 += lhs[ I(j,2) ] * rhs_ij;
      ri3 += lhs[ I(j,3) ] * rhs_ij;
    }
```

```
        r[ I(i,0) ] = ri0;
        r[ I(i,1) ] = ri1;
        r[ I(i,2) ] = ri2;
        r[ I(i,3) ] = ri3;
    }
}
```

Результати тестів продуктивності вбудованого рішення по визначеному алгоритму тестування на ідентичному наборі вихідних даних та за ідентичного апаратного та програмного середовища (табл. 2, рис. 3):

Таблиця 2. Результати тестування продуктивності нативного вбудованого підходу

Ітерація, №	Затрати процесорного часу, мс
1	6075
2	5997
3	5987
4	6008
5	6037
6	6023
7	6005
8	5994
9	5973
10	6053
Усереднений результат	6015

Виходячи з приведених результатів можливо зробити висновок, що в даному випадку нативна реалізація даної задачі, що поставляється в пакеті Android, демонструє результати в середньому у 1,9 рази повільніші ніж наївна реалізація того ж алгоритму в просторі JDK.

Можливо виділити дві основні причини такої низької продуктивності даної реалізації. Перша – наявність накладних витрат виклику нативних функцій за використанням JNI, так званий «overhead» [16]. Друга – наявність цілого ряду додаткових перевірок вхідних даних та надлишкового копіювання даних у тимчасові буфери для проведення обчислень.

Виходячи з проведених тестів, націлених на оцінку витрат виклику JNI за використанням Google Caliper можна зробити висновок, що на цільовому апаратно-програмному забезпеченні виклик JNI функцій може займати в перспективі час еквівалентний 5-30 Java операціям над примітивними типами (табл. 3) [17, 18]:

Scenario JniCall}

10.26 ns; σ=0.02 ns @ 10 trials

Scenario{AddIntOperation}

0.48 ns; σ=0.02 ns @ 10 trials

Scenario{AddLongOperation}

0.87 ns; σ=0.02 ns @ 10 trials

Таблиця 3. Результати тестування накладних витрат виклику нативних функцій та Java-операцій над примітивними типами

Тест	Витрати процесорного часу, нс
JniCall	10.265
AddIntOperation	0.481
AddLongOperation	0.873

Таким чином, можливо запропонувати вирішення даної задачі прибираючи максимальну кількість надлишкових перевірок і вбудованих методів захисту і перевірки нативної імплементації, що допустимо на такому низькому рівні виконання, оскільки предметна область даного рішення передбачає використання у складі бойових, відлагоджених і стабільних систем, де в принципі не допустима передача не дійсних вхідних параметрів для обчислень, оскільки відповідальність за такі перевірки та препроцесінг у таких системах несуть верхні рівні, абстракції та реалізації системи.

Також можливо оптимізувати представлений наївний алгоритм добутку матриць введенням константних індексів, агресивних рівнів оптимізації компілятора нативного рішення.

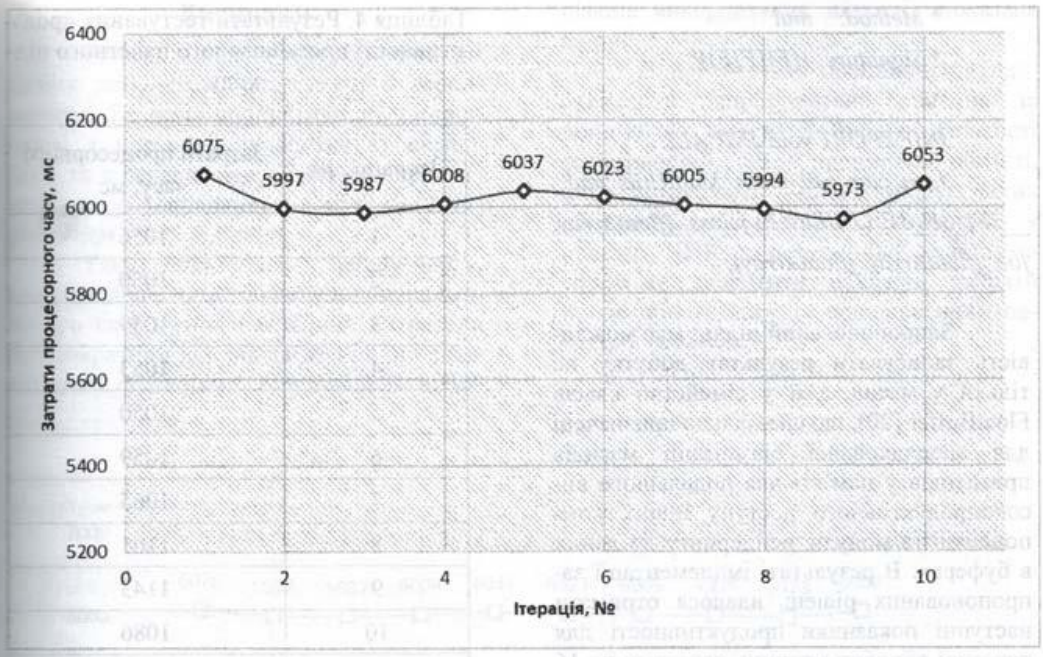


Рис. 3. Графік залежності затрат процесорного часу від ітерації тестування для нативного вбудованого підходу

Окрім цього, в контексті цих самих систем у більшості випадків існує потреба саме пакетної обробки даних. Маючи можливість отримувати результат добутку десятків, сотень матриць одночасно можливо, відповідно, багатократно скоротити накладні затрати на виклик високопродуктивних JNI функцій. Цей підхід, безперечно, потребує важкої та високопродуктивної імплементації на рівні клієнтського коду JVM, проте цілком заслуговує його.

Для імплементації запропонованого рішення використовується відносно нова підтримка роботи з NDK в складі середи розробки Android Studio. За допомогою цього інструментарію буде згенеровано скрипт ndk-build для генерації бінарних файлів, визначення правил компіляції.

Файл Application.mk [19]:

```
APP_ABI := all
APP_PLATFORM := android-16
APP_CPPFLAGS += -std=c++11
```

У даному файлі конфігурації було визначено:

підтримку генерації машинного коду для всіх підтримуваних процесорних архітектур, а саме: Hardware FPU instructions on ARMv7 based devices, ARMv8 AArch64, IA-32, Intel64, MIPS32, MIPS64 (r6), мінімальну підтримувану версію Android-платформи, підтримку стандарту C++11.

Header-файл:

```
/*
 * Method:    mul
 * Signature: (LBuffer;I[F[F)V
 */
JNIEXPORT void JNICALL
Java_com_util_math_MathUtils_mul_
_Ljava_nio_Buffer_2I_3F_3F  (JNIEnv  *,
jclass, jobject, jint, jfloatArray, jfloatArray);
/*
```

```
* Method:    mul
* Signature: ([FI[F[F)V
*/
JNIEXPORT void JNICALL
Java_com_util_math_MathUtils_mul_
__3FI_3F_3F (JNIEnv *, jclass, jfloatArray,
jint, jfloatArray, jfloatArray);
```

Запропонований підхід має можливість записувати результат добутку не тільки у масив, а й у сімейство класів FloatBuffer [20], що спеціально призначені для впорядкованої організації масивів примітивів у пам'яті для подальшого високопродуктивного доступу інших компонентів та модулів рендерингу до даних в буферах. В результаті імплементації запропонованих рішень вдалося отримати наступні показники продуктивності для пакетної обробки матриць пакетами по 16 [21] у перерахунку на мільйон матриць (табл. 4, рис. 4):

Таблиця 4. Результати тестування продуктивності пропонованого пакетного підходу

Ітерація, №	Затрати процесорного часу, мс
1	1056
2	1050
3	1059
4	1053
5	1050
6	1059
7	1065
8	1101
9	1143
10	1086
Усереднений результат	1072

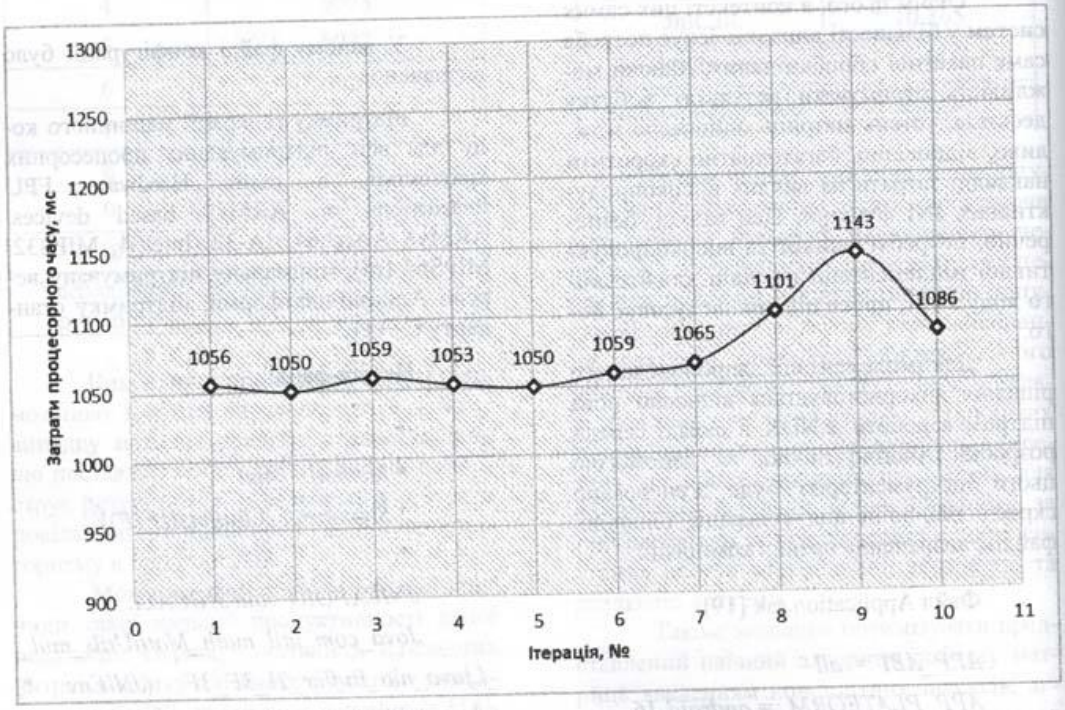


Рис. 4. Графік залежності затрат процесорного часу від ітерації тестування для пропонованого пакетного підходу

Висновки

Таким чином, запропоноване рішення демонструє результати в середньому у 5,6 разів продуктивніші у порівнянні з нативним рішенням із складу SDK та в середньому у 2,9 разів продуктивніші у порівнянні з наївним Java-рішенням.

Таких показників, у першу чергу, вдалося досягнути за рахунок максимального скорочення накладних витрат на багатократний виклик та взаємодію з нативними функціями. Порівняльна візуалізація використаних підходів показана на рис. 5.

В перспективі пропонується вдосконалити запропоноване рішення за допомогою інтеграції більшої кількості особливостей систем предметної області, що потребують високопродуктивних обчислень на цільовій платформі, у простір NDK з попередньо вказаними спробами скоротити накладні витрати процесорного часу та провадженням пакетних обробок.

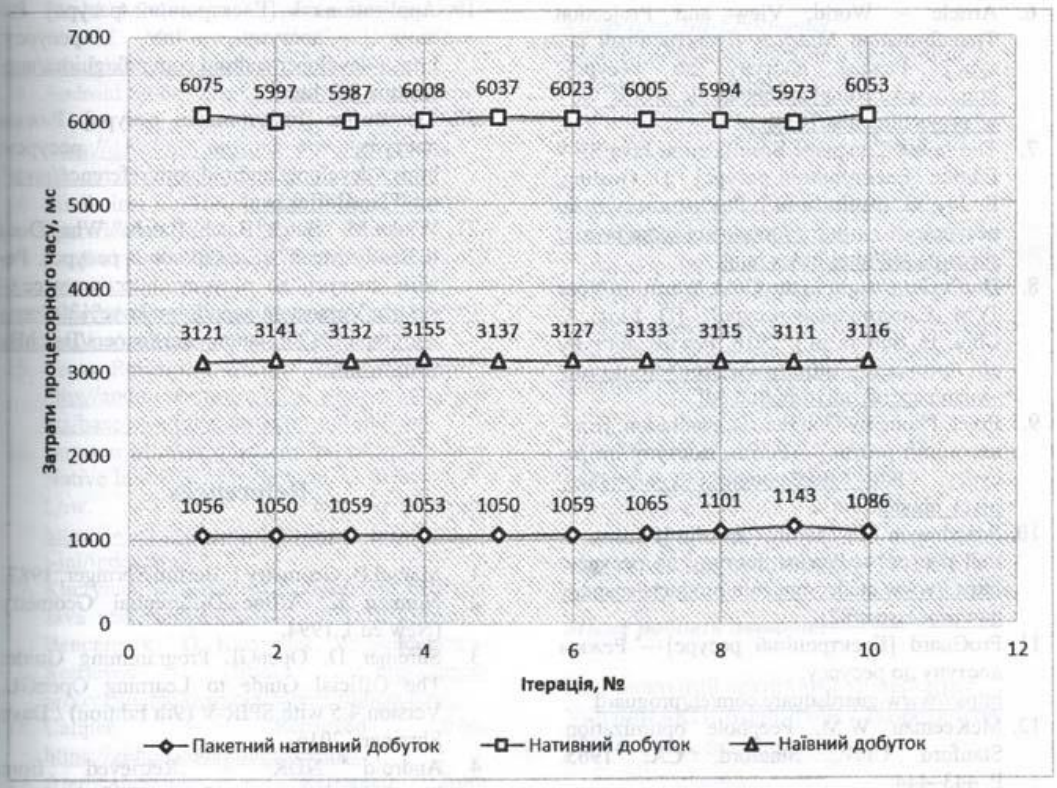


Рис. 5. Графік залежності затрат процесорного часу від ітерації тестування для всіх розібраних та запропонованих підходів

Література

1. Marcel B. Geometry I / Berger Marcel. – Berlin: Springer, 1987.
2. Nomizu K. Affine Differential Geometry (New ed.) / K. Nomizu, S. S., 1994.
3. Shreiner D. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V (9th Edition) / Dave Shreiner., 2016.
4. Android NDK [Електронний ресурс]. Режим доступу до ресурсу: https://developer.android.com/ndk/index.html.
5. Java Native Interface Specification [Електронний ресурс]. Режим доступу до ресурсу: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html.
6. Article – World, View and Projection Transformation Matrices [Електронный ресурс]. Режим доступу до ресурсу: http://www.codinglabs.net/article_world_view_projection_matrix.aspx.
7. The Java® Language Specification Java SE 9 Edition [Електронний ресурс] / [J. Gosling, B. Joy, G. Steele та ін.]. Режим доступу до ресурсу: https://docs.oracle.com/javase/specs/jls/se9/html/index.html.
8. Don't Get Caught In the Cold, Warm-up Your JVM [Електронный ресурс] / [D. Lion, A. Chiu, H. Sun та ін.]. 2016. Режим доступу до ресурсу: http://www.eecg.toronto.edu/~yuan/papers/osdi16-hottub.pdf.
9. Pixel, Phone by Google. 1st generation. [Електронний ресурс] – Режим доступу до ресурсу: https://store.google.com/gb/product/pixel_phone.
10. Snapdragon 821 Mobile Platform [Електронний ресурс] – Режим доступу до ресурсу: https://www.qualcomm.com/products/snapdragon/processors/821.
11. ProGuard [Електронний ресурс] – Режим доступу до ресурсу: https://www.guardsquare.com/en/proguard.
12. McKeeman W.M. Peephole optimization. Stanford Univ., Stanford, CA. 1965. P. 443–444.
13. Android Studio The Official IDE for Android [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/studio/index.html.
14. Lindholm T., Yellin F., Bracha G. The Java® Virtual Machine Specification Java SE 9 Edition [Електронний ресурс]. Режим доступу до ресурсу: https://docs.oracle.com/javase/specs/jvms/se9/html/index.html.
15. Cross Reference: util.cpp [Електронний ресурс]. Режим доступу до ресурсу: http://androidxref.com/8.0.0_r4/xref/frameworks/base/core/jni/android/opengl/util.cpp.
16. Dawson M., Johnson G., Low A. Best practices for using the Java Native Interface [Електронний ресурс]. 2009. Режим доступу до ресурсу: https://www.ibm.com/developerworks/library/j-jni/index.html.
17. Kurzyniec D., Sunderam V. Efficient Cooperation between Java and Native Codes – JNI Performance Benchmark [Електронний ресурс]. Режим доступу до ресурсу: http://janet-project.sourceforge.net/papers/jnibench.pdf.
18. Caliper [Електронний ресурс]. Режим доступу до ресурсу: https://github.com/google/caliper.
19. Application.mk [Електронний ресурс]. Режим доступу до ресурсу: https://developer.android.com/ndk/guides/application_mk.html.
20. FloatBuffer [Електронний ресурс]. Режим доступу до ресурсу: https://developer.android.com/reference/java/nio/FloatBuffer.html.
21. Wloka M. "Batch, Batch, Batch:" What Does It Really Mean? [Електронний ресурс]. Режим доступу до ресурсу: http://www.ce.u-sys.org/Veranstaltungen/Interaktive%20Computergraphik%20(Stamminger)/papers/BatchBatchBatch.pdf.

References

1. Marcel B. Geometry I. Berlin: Springer, 1987.
2. Nomizu K. Affine Differential Geometry (New ed.), 1994.
3. Shreiner D. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V (9th Edition) / Dave Shreiner., 2016.
4. Android NDK – Retrieved from https://developer.android.com/ndk/index.html.
5. Java Native Interface Specification – Retrieved from https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html.
6. Article - World, View and Projection Transformation Matrices – Retrieved from http://www.codinglabs.net/article_world_view_projection_matrix.aspx.
7. The Java® Language Specification Java SE 9 Edition / [J. Gosling, B. Joy, G. Steele та ін.]

– Retrieved from https://docs.oracle.com/javase/specs/jls/se9/html/index.html.
8. Don't Get Caught In the Cold, Warm-up Your JVM / [D. Lion, A. Chiu, H. Sun та ін.]. – 2016. – Retrieved from http://www.eecg.toronto.edu/~yuan/papers/osdi16-hottub.pdf.
9. Pixel, Phone by Google. 1st generation. – Retrieved from https://store.google.com/gb/product/pixel_phone.
10. Snapdragon 821 Mobile Platform – Retrieved from https://www.qualcomm.com/products/snapdragon/processors/821.
11. ProGuard – Retrieved from https://www.guardsquare.com/en/proguard.
12. W. M. McKeeman. Peephole optimization / W. M. McKeeman. // Stanford Univ., Stanford, CA. – 1965. – C. 443–444.
13. Android Studio The Official IDE for Android – Retrieved from https://developer.android.com/studio/index.html.
14. Lindholm T. The Java® Virtual Machine Specification Java SE 9 Edition / T. Lindholm, F. Yellin, G. Bracha – Retrieved from https://docs.oracle.com/javase/specs/jvms/se9/html/index.html.
15. Cross Reference: util.cpp – Retrieved from http://androidxref.com/8.0.0_r4/xref/frameworks/base/core/jni/android/opengl/util.cpp.
16. Dawson M. Best practices for using the Java Native Interface / M. Dawson, G. Johnson, A. Low. – 2009. – Retrieved from https://www.ibm.com/developerworks/library/j-jni/index.html.
17. Kurzyniec D. Efficient Cooperation between Java and Native Codes – JNI Performance Benchmark / D. Kurzyniec, V. Sunderam – Retrieved from http://janet-project.sourceforge.net/papers/jnibench.pdf.
18. Caliper – Retrieved from https://github.com/google/caliper.
19. Application.mk – Retrieved from https://developer.android.com/ndk/guides/application_mk.html.
20. FloatBuffer – Retrieved from https://developer.android.com/reference/java/nio/FloatBuffer.html.
21. Wloka M. "Batch, Batch, Batch:" What Does It Really Mean? / Matthias Wloka – Retrieved from http://www.ce.u-sys.org/Veranstaltungen/Interaktive%20Computergraphik%20(Stamminger)/papers/BatchBatchBatch.pdf.

Одержано 05.12.2017

***Про авторів*:**

Ашур Ілля Зін-Еддінович,
магістрант, Національний технічний університет України
«КПІ імені Ігоря Сікорського»
https://orcid.org/0000-0003-2348-8777

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук, професор, завідувач відділу теорії комп'ютерних обчислень Інституту програмних систем НАН України, професор кафедри автоматики і управління в технічних системах НТУУ "КПІ".
Кількість наукових публікацій в українських виданнях – понад 150.
Кількість наукових публікацій в іноземних виданнях – понад 50.
Індекс Гірша – 5.
http://orcid.org/0000-0002-8435-1451.

Місце роботи авторів:

Національний технічний університет України «КПІ імені Ігоря Сікорського»
03056, Київ, пр. Перемоги, 37

E-mail: ilyaachour@gmail.com,
doroshenkoanatoliy2@gmail.com

ДОДАТОК Б

Институт проблем моделирования в энергетике им. Г.Е. Пухова НАН Украины
НИИ многопроцессорных вычислительных систем им А.В. Каляева Южного Федерального Университета, Россия
Национальный технический университет «Львовская политехника»
Институт кибернетики им.В.М. Глушкова НАН Украины
Национальный авиационный университет Украины
Донецкий национальный технический университет
Институт электродинамики НАН Украины
Институт проблем регистрации информации НАН Украины
Щецинский технический университет, Польша
Институт специальной связи и защиты информации НТУУ «КПИ»
Ташкентский государственный технический университет, Узбекистан
Компания «Юстар», Украина
Энергосервисная компания «ПАТРИОТ-НРГ»

Сборник трудов конференции

МОДЕЛИРОВАНИЕ-2016

SIMULATION-2016

25-27 мая 2016, Киев

УДК 004.94

М74

Институт проблем моделирования в энергетике им. Г.Е. Пухова НАН Украины, НИИ многопроцессорных вычислительных систем им А.В. Каляева Южного Федерального Университета (Россия), Национальный технический университет «Львовская политехника», Институт кибернетики им.В.М. Глушкова НАН Украины, Национальный авиационный университет Украины, Донецкий национальный технический университет, Институт электродинамики НАН Украины, Институт проблем регистрации информации НАН Украины, Щецинский технический университет (Польша), Институт специальной связи и защиты информации НТУУ «КПИ», Ташкентский государственный технический университет (Узбекистан), Компания «Юстар» (Украина), Энергосервисная компания «ПАТРИОТ-НРГ» проводят Пятую международную конференцию МОДЕЛИРОВАНИЕ-2016.

Конференция проходит в Институте проблем моделирования в энергетике им. Г.Е. Пухова НАН Украины, г. Киев, 25-27 мая 2016 г.

ISBN 978-966-02-7928-5

Все материалы поданы в авторской редакции. Прошли рецензирование.

И. З. Ашур, бакалавр, системная инженерия,
Национальный технический университет Украины «КПИ»
(Украина, 03056, Киев, ул. Борщаговская, 126,
тел. +380972938453, e-mail: ilyaachour@gmail.com)

Система 2D рендеринга с использованием интерфейса OpenGL ES 2.0, особенностей и техник VBO/Sprite Batch

This work presents high-performance graphics engine implementation based on OpenGL ES for Android OS devices. Approach performance was precisely measured, all used features and technics combinations were considered and benchmarked.

Ключевые слова: OpenGL ES 2.0, VBO, Sprite Batch, Android, Java

Введение. Разработка приложений для мобильных платформ, использующих высокопроизводительную двумерную компьютерную графику по спецификации и на программном интерфейсе OpenGL требует создания комплексной архитектуры для каждого отдельного решения.

Такие приложения разрабатываются для игровой индустрии, программного обеспечения визуализации, моделирования в части рендеринга визуальных представлений элементов моделей, визуального интерактивного взаимодействия.

Подмножество графического интерфейса OpenGL ES[1)] (OpenGL for Embedded Systems — OpenGL для встраиваемых систем) предназначено для встраиваемых систем — мобильных телефонов, умных часов, телевизоров и т.д.

Цель работы. Проводимое исследование нацелено на разработку, создание и анализ производительности универсальной и максимально простой объектно-ориентированной программной прослойки двумерного графического движка на языке программирования Java/C++(JNI) под платформу Android версии 4.1+ (API Level 16) с определенным, расширенным функционалом.

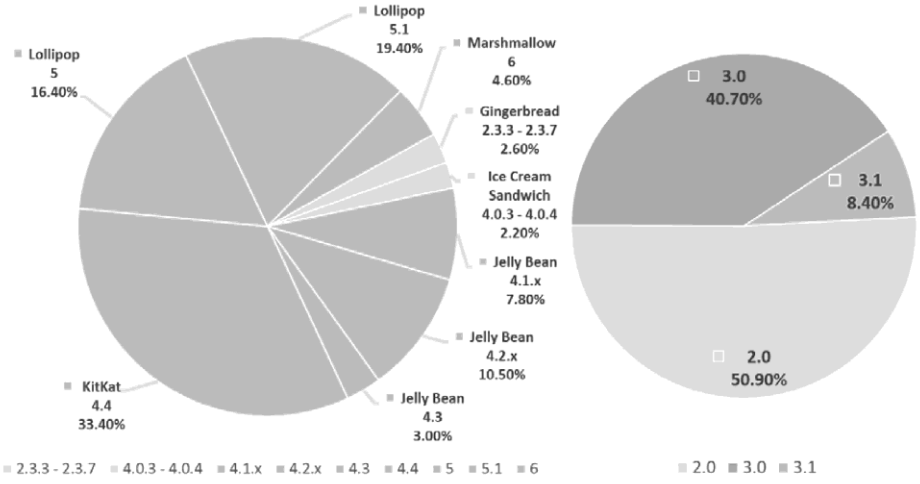


Рис. 2. Сравнительный анализ количества устройств, поддерживающих различные версии OpenGL ES[2)] на базе различных версий Android[3)] по состоянию на 4 апреля 2016 года

Решение заключается в инкапсуляции использования таких сложных общих практик, механизмов, техник и особенностей рендеринга[4] OpenGL ES, как VBO[5], Sprite Batching, Buffer Orphaning[6], Object Pooling, Caching, Boxing avoiding, Alpha Blending, Color Blending, Transparency Sorting[7], Thread Safety, Native Buffer Operations и т.д. в целях достижения максимальной производительности на ряду с простотой использования, гибкостью и функциональностью прослойки.

Помимо внутреннего, инкапсулированного функционала необходимо создать ряд интерфейсов и методов для осуществления базовых операций с графическими элементами так называемого “движка”. К таким операциям относятся поддержка матриц аффинных преобразований (матриц модели и вида), обрезка, управление прозрачностью и цветом, упрощенные операции трансформации графических элементов (масштабирование, поворот, перемещение), управление их видимостью, создание, воспроизведение и управление анимацией.

Одна из главных особенностей разработанной прослойки — использование Vertex Buffer Object и матриц модели-вида-проекции с кэшируемым содержанием, что позволяет одновременно и единожды выгружать информацию о вершинах, текстурах и прочих параметрах группы графических элементов (Sprite Batching) в видеоустройство, что эффективно увеличивает производительность рендеринга, особенно касаемо статической графики (графики с неизменяемыми или редко изменяемыми координатами позиций вершин и их текстур).

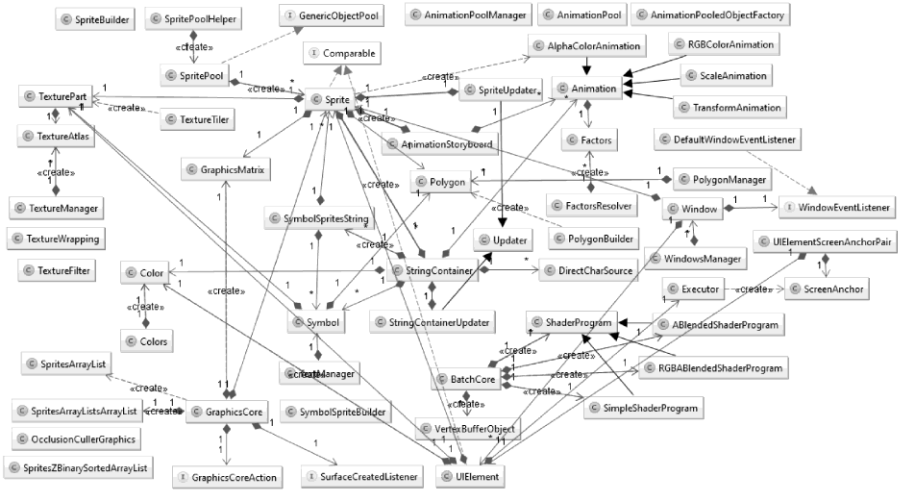


Рис. 3. Схема классов и зависимостей разработанного решения в части графического ядра

Оценка. Для оценки производительности решения были созданы специальные программы-банчмарки с учетом всех “best practices”. Написание корректных тестов производительности для определенной части большого приложения — сложная задача. Существует множество оптимизаций, которые JVM или аппаратное обеспечение могут применить к конкретному компоненту во время его изолированного тестирования. Непродуманные тесты производительности могут дать оптимистичные результаты, отнюдь не совпадающие с реальными. Именно тестирование конкретного компонента в составе рабочего, практичного

приложения с применением таких техник, как JVM warm up (“разогрев” виртуальной машины) может дать достоверные результаты.

Далее будут приведены результаты тестов производительности для итераций разработанного компонента с различающимся стеком используемых техник и особенностей, их сравнение с производительностью реализаций подобных сторонних компонентов. Под метрикой производительности подразумевается время, затраченное процессором и видеоядром на рендеринг одного кадра смешанного (статического и динамического) содержания. Для бенчмаркинга видеоядра использовались нативные инструменты производителей — такие, как: Intel GPA, Mali Developer Tools, PowerVR Tools, Snapdragon Profiler, Adreno GPU Profiler, DS Development Studio.

Сцена для тестов содержит 2000 однотипных спрайтов. Тесты для разного количества спрайтов опускаются по причине практически прямой зависимости между количеством последних и производительностью рендеринга.

Представленные результаты получены на устройстве с процессором Quad-core 2.3 GHz Krait 400 и графическим ядром Adreno 330.

N	VBO Type	VBO Cache	VBO indicies	Matrices processing	Matrices cache
1	Instanced VBO	No cache	No indicies	GPU-side matrices processing	No cache
2	Batched VBO	No cache	No indicies	GPU-side matrices processing	No cache
3	Batched VBO	Client-side cache, managed uploading	No indicies	GPU-side matrices processing	No cache
4	Batched VBO	Client-side cache, managed uploading	Immediate VBO indicies uploading	GPU-side matrices processing	No cache
5	Batched VBO	Client-side cache, managed uploading	Immediate VBO indicies uploading	CPU-side matrices processing	No cache
6	Batched VBO	Client-side cache, managed uploading	One-time VBO indicies uploading	CPU-side matrices processing	No cache
7	Batched VBO	Client-side cache, managed uploading	One-time VBO indicies uploading	CPU-side matrices processing	Client-side cache
8	Batched VBO	Client-side cache, native uploading	One-time VBO indicies uploading	CPU-side matrices processing	Client-side cache
9	Batched orphaned VBO	Client-side cache, native uploading	One-time VBO indicies uploading	CPU-side matrices processing	Client-side cache

Рис. 4. Итерации разработки компонента на основе внедрения техник и особенностей. Градации серого отражают относительную производительность данной особенности или техники от темного к светлому

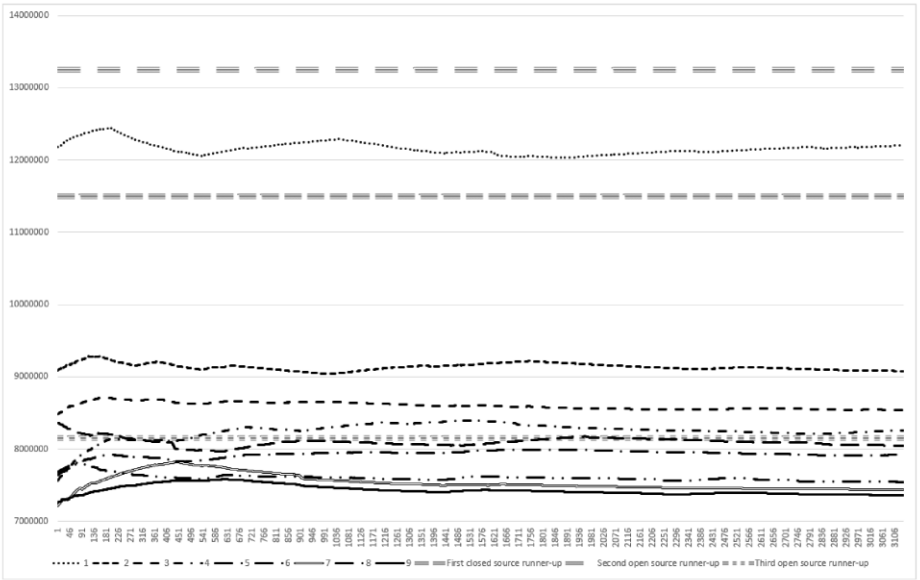


Рис. 5. Графики производительности итераций и решений. Представлено время рендеринга одного кадра в наносекундах

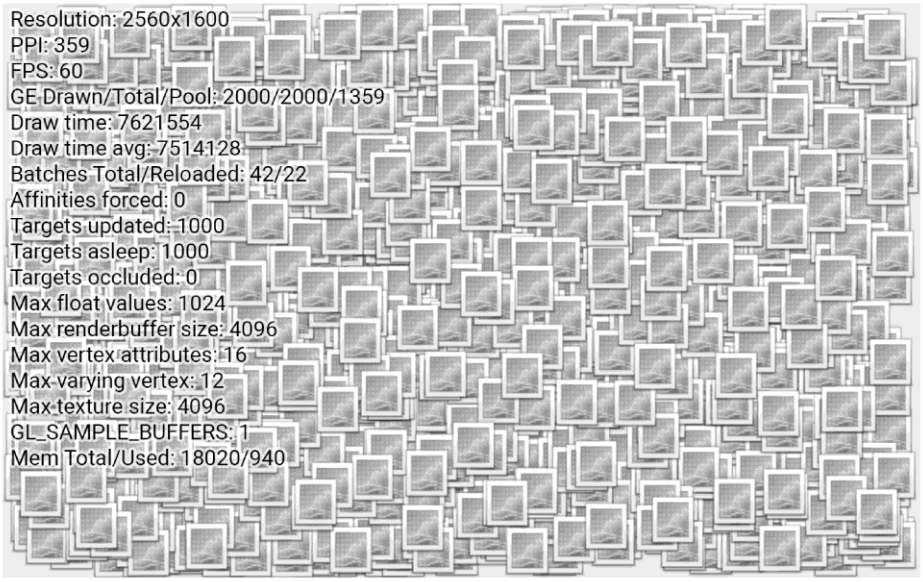


Рис. 6. Пример тестовой сцены с отладочной информацией[8)]

Заключение. Предложено решение для производительного аппаратно-ускоренного рендеринга на устройствах под управлением ОС Android, основанное на ряде сложных общих практик, механизмов, техник и особенностей.

Использование данного решения позволяет максимально просто и быстро визуализировать любую высоконагруженную двумерную графику.

Тесты производительности показали, что решение позволяет визуализировать большие объемы графики на примере 2000 спрайтов за промежутки времени до 7.5 миллисекунд, что составляет всего 45% от времени, отведенного исследуемой выборкой устройств на рендеринг одного кадра по соображениям вертикальной синхронизации[9)].

Дальнейшее усовершенствование предусматривает внедрение новых техник и особенностей OpenGL ES последующих версий.

1. OpenGL ES [Электронный ресурс] // Режим доступа: https://www.khronos.org/opengles/
2. Dashboards [Электронный ресурс] // Режим доступа: http://developer.android.com/intl/ru/about/dashboards/index.html#OpenGL
3. Dashboards [Электронный ресурс] // Режим доступа: http://developer.android.com/intl/ru/about/dashboards/index.html#Platform
4. Vertex Rendering [Электронный ресурс] // Режим доступа: https://www.opengl.org/wiki/Vertex_Rendering
5. Vertex Specification [Электронный ресурс] // Режим доступа: https://www.opengl.org/wiki/Vertex_Specification
6. Buffer Object Streaming [Электронный ресурс] // Режим доступа: https://www.opengl.org/wiki/Buffer_Object_Streaming
7. Transparency Sorting [Электронный ресурс] // Режим доступа: https://www.opengl.org/wiki/Transparency_Sorting
8. Icons [Электронный ресурс] // Режим доступа: http://p.yusukekamiyamane.com/
9. Implementing graphics [Электронный ресурс] // Режим доступа: https://source.android.com/devices/graphics/implement.html#vsync

ДОДАТОК В

CONFERENCE PROCEEDINGS

МАТЕРІАЛИ КОНФЕРЕНЦІЇ

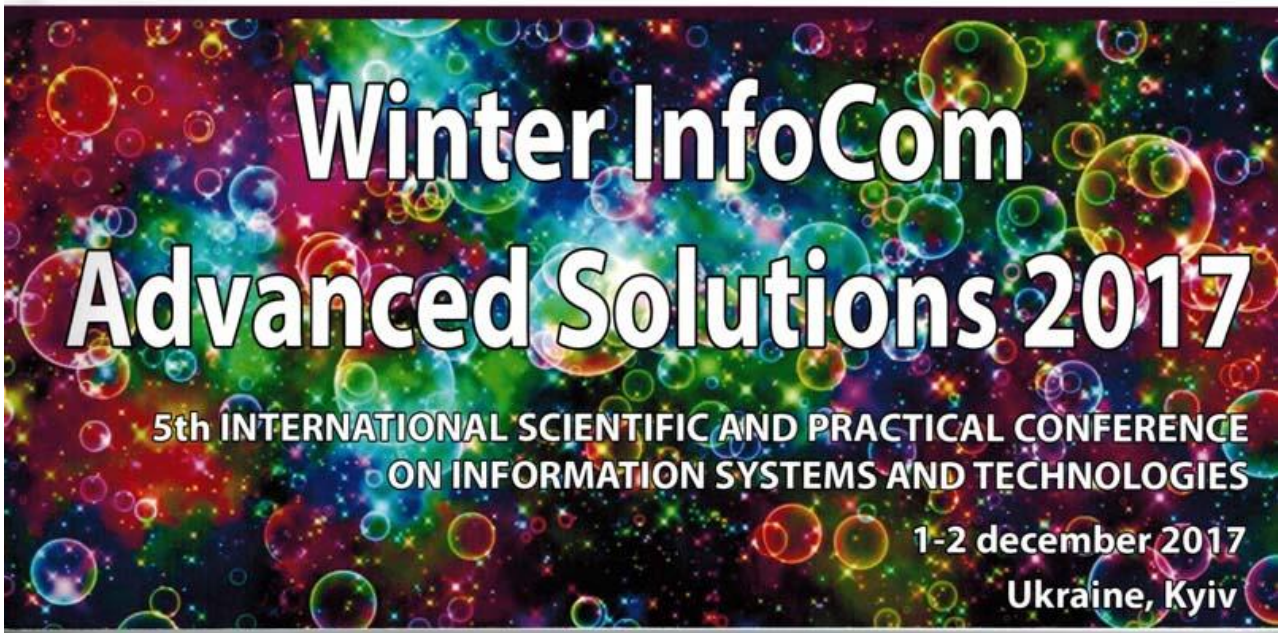
winter infoCom 2017

Winter InfoCom Advanced Solutions 2017

V МІЖНАРОДНА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ
З ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

1-2 грудня 2017 року
Україна, Київ

ISBN 978-966-2344-58-5



МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНСТИТУТ МОДЕРНІЗАЦІЇ ЗМІСТУ ОСВІТИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

WINTER INFOCOM ADVANCED SOLUTIONS 2017

МАТЕРІАЛИ

V МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ

CONFERENCE PROCEEDINGS

5th SCIENTIFIC AND PRACTICAL CONFERENCE

КИЇВ, УКРАЇНА

1-2 ГРУДНЯ 2017 РОКУ

УДК 004

Winter InfoCom 2017: Матеріали V Міжнародної науково-практичної конференції з інформаційних систем та технологій, м. Київ, 1-2 грудня 2017 р. – К.: Вид-во ТОВ "Інжиніринг", 2017. – 76 с. – Мови укр., рос., англ.

Конференція входить до Переліку міжнародних та всеукраїнських науково-практичних конференцій здобувачів вищої освіти та молодих учених у 2017 році (додаток до листа Міністерства освіти і науки України № 1/9-24 від 23 січня 2017 року).

Проведення конференції регламентоване наказом ректора КПІ ім. Ігоря Сікорського № 3-469 від 06 листопада 2017 р.

Редакційна колегія дотримується прийнятих міжнародною спільнотою принципів публікаційної етики, відображених, зокрема, в рекомендаціях Комітету з етики наукових публікацій (Committee on Publication Ethics, COPE), а також враховує досвід авторитетних міжнародних видавництв. Щоб уникнути недобросовісної практики в публікаційній діяльності (плагіат, виклад недостовірних відомостей та ін.), з метою забезпечення високої якості наукових публікацій, визнання громадськістю отриманих автором наукових результатів, кожен член редакційної колегії, автор, рецензент, видавець, а також установи, які беруть участь в видавничому процесі, зобов'язані дотримуватися етичних стандартів, норм і правил та вживати всіх можливих заходів для запобігання їх порушень. Дотримання правил етики наукових публікацій усіма учасниками цього процесу сприяє забезпеченню прав авторів на інтелектуальну власність, підвищенню якості видання і виключення можливості неправомірного використання авторських матеріалів в інтересах окремих осіб.

ISBN 978-966-2344-58-5

ПРОГРАМА/PROGRAM

1 грудня

Інформаційні системи та технології	
Карымсакова И.Б. **Денисова Н.Ф.** **Крак Ю.В.**	Разработка роботизированной системы для плазменного напыления имплантов
Дорошенко А.Ю. **Туманов В.В.**	Модуль калібрування та позиціонування системи комп'ютерного зору для цифрової нарізки матеріалів
Коноваленко А.	Система для організації інтерактивних квест-ігор побудована на Bluetooth-маячках
Системи керування	
Юрчук Л.Ю. **Бублінський С.М.**	Проектування відеокадрів людино-машинного інтерфейсу АСУ ТП
Лапханов Э.А.	Оценка возможности создания дополнительной тяги для управления космическими аппаратами на основе использования постоянных магнитов
Технології програмування	
Ашур И.З. **Дорошенко А.Ю.**	Высокопроизводительное произведение матриц посредством Android NDK и JNI
Оброблення інформації у складних системах	
Дмитренко О.О. **Ланде Д.В.**	Метод накопичувального впливу для аналізу когнітивних карт
Підготовка кадрів у галузі інформаційних технологій	
Стенин А.А. **Пасько В.П.** **Лемешко В.А.** **Шитикова И.Г.**	Оптимизация учебных планов IT-специальностей

ЗМІСТ/CONTENTS

Высокопроизводительное произведение матриц посредством Android NDK и JNI

Ашур И.З.
КПИ им. Игоря Сикорского
Киев, Украина
ilyaachour@gmail.com

Дорошенко А.Ю.
КПИ им. Игоря Сикорского
Киев, Украина
a-y-doroshenko@ukr.net

Аннотация. **Работа представляет описание высокопроизводительного подхода для произведения матриц посредством Android NDK и JNI. Была произведена оценка производительности подхода и его альтернатив, рассмотрены все используемые техники и особенности.**

Ключевые слова: Android NDK, JNI, аффыиные преобразования, OpenGL ES, Java

Введение

Эффективное произведение матриц – одна из актуальных проблем разработки приложений для мобильных платформ, использующих высокопроизводительную двумерную компьютерную графику.

Такие приложения разрабатываются для игровой индустрии, программного обеспечения визуализации, моделирования в части рендеринга визуальных представлений элементов моделей, визуального интерактивного взаимодействия.

Одним из аспектов применения результатов данных вычислений является произведения матриц вида, модели и проекции.

Матричное представление используется, в частности, для записи аффинных преобразований в компьютерной графике.

Анализ предметной области

Матрица вида описывает представление аффинного преобразования, например, сжатие, растяжение, поворот, параллельный перенос, отражение и другие частные и общие случаи трансформаций.

Пространство вида представляет результат преобразования мировых координат в координаты видимого пространства. Видимое пространство в данном случае определяется совокупностью преобразований сдвигов и вращений сцены. Эти комбинированные преобразования представляются матрицей вида.

Современные системы рендеринга двумерной графики также оперируют понятием пространства отсечения. Пространство отсечения определяет видимые области сцены и, соответственно, вершины.

Общий процесс применений данных преобразований приведен на рис. 1 [1].

Детали применения данных трансформаций и математическая их составляющая не является предметом обсуждения данной статьи и упомянуты с целью предоставления общей информации о предметной области решаемой задачи.

Проводимое исследование нацелено на разработку, создание и анализ производительности подхода произведения матриц посредством Android NDK и JNI.

Анализ и реализация решений

Отталкиваясь от введенных ограничений предметной области в виде матриц аффинных преобразований, в дальнейшем будем оперировать матрицами размером 4×4.

Классическая, наивная реализация произведения матриц предусматривает использование средств JDK с применением двух вложенных циклов.

Для оценки производительности подхода и его наивных аналогов были созданы специальные программы-бенчмарки с учетом всех «best practices». Существует множество оптимизаций, которые JVM или аппаратное обеспечение могут применить к конкретному компоненту во время его изолированного тестирования. Непродуманные тесты производительности могут дать оптимистичные результаты, отнюдь не совпадающие с реальными. Именно тестирование конкретного компонента в составе рабочего, практичного приложения с применением таких техник, как JVM warm up («разогрев» виртуальной машины) может дать достоверные результаты [2].

Рис. 1. Общая схема применения матриц

Далее будут приведены результаты тестов производительности для итераций разработанного подхода и его наивных альтернатив с различающимся стеком используемых техник и особенностей, их сравнение. Под метрикой производительности подразумевается время, затраченное процессором на выполнение определенного количества матричных произведений.

Результаты тестирование затрат процессорного времени на произведение 1 миллиона матриц 4×4 в представлении одномерного массива на 16 элементов типа с плавающей запятой с использованием классической, наивной реализации средствами JDK на устройстве Google Pixel XL в 10 итераций представлены в табл. 1 и рис. 2.

Таблица 1

Итерация, №	Затраты процессорного времени, мс
1	3121
2	3141
3	3132
4	3155
5	3137
6	3127
7	3133
8	3115
9	3111
10	3116
Усредненный результат	3129

Результаты тестов для разного количества матриц, разного количества итераций, разных устройств и разной размерности матриц опускаются по причине практически прямой зависимости между приведенными метриками и производительностью вычислений.

Представленные результаты получены на устройстве с процессором Qualcomm MSM8996 Snapdragon 821 Quad-core (2x2.15 GHz Kryo & 2x1.6 GHz Kryo) [3].

Следует отметить, что тесты производительности производились на сборках приложения с использованием агрессивной оптимизации и обфускации исходного кода посредством инструмента ProGuard с семью проходами оптимизации, применяющих, в частности, арифметические

Рис. 2. График зависимости затрат процессорного времени от итерации тестирования для наивной реализации

оптимизации, финализацию классов, их членов, вертикальное и горизонтальное склеивание классов, приватизации полей, методов, применение статичных методов, оптимизация количества параметров методов, встраивание константных параметров, встраивание констант возврата методов, встраивание вызовов коротких методов, встраивание уникальных методов, набор методик «peephole optimization» для арифметических операций, для ветвления кода, для строк, оптимизации приведения типов, чистка избыточных инструкций, исключений, блоков кода [4]. Тема оптимизаций производительности Java-кода на этапе компиляции проекта на текущий момент достаточно изучена и глубоко разработана, документирована известными авторитетными разработчиками и не нуждается в более подробном рассмотрении. Данные оптимизации применялись для получения «боевых» результатов тестирования производительности.

В составе Android SDK предоставляется набор классов-утилит для работы с матрицами. Их реализация, подобно далее предложенному подходу, также имплементирована на уровне NDK [5]. В целом, она полностью повторяет рассмотренный выше подход за исключением наличия целого ряда дополнительных защит и проверок, инструкций для высвобождения памяти.

Результаты тестов производительности встроенного решения по отработанному алгоритму тестирования на идентичном наборе исходных данных и при идентичном окружении аппаратной и программной среды представлены на табл. 2 и рис. 3:

Таблица 2

Итерация, №	Затраты процессорного времени, мс
1	6075
2	5997
3	5987
4	6008
5	6037

6	6023
7	6005
8	5994
9	5973
10	6053
Усредненный результат	6015

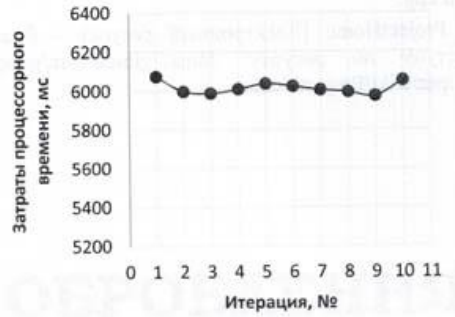


Рис. 3. График зависимости затрат процессорного времени от итерации тестирования для встроенной реализации

Исходя из приведенных результатов можно сделать вывод, что в данном случае нативная реализация данной задачи, поставляемая в составе пакета разработки Android, показывает результаты в среднем в 1,9 раз медленнее, чем наивная реализация того же алгоритма в пространстве JDK.

Возможно выделить две основные причины такой низкой производительности данной реализации. Первая – накладные затраты вызова нативных функций посредством JNI, т.н. «overhead». Вторая – наличие целого ряда дополнительных проверок входных данных и избыточного копирования данных во временные буферы для проведения вычислений.

Исходя из проведенных тестов, нацеленных на оценку затрат вызова JNI функций посредством Google Caliper [6] (рис. 4), можно сделать вывод, что на целевом аппаратном обеспечении вызов JNI функции может занимать перспективно время эквивалентное выполнению 5-30 java операций над примитивными типами.

```
Scenario{benchmark=Jni} 10.26 ns; σ=0.02 ns @ 10 trials
Scenario{benchmark=IntOperation} 0.48 ns; σ=0.02 ns @ 10 trials
Scenario{benchmark=LongOperation} 0.87 ns; σ=0.02 ns @ 10 trials

     benchmark     ns linear runtime
           Jni 10.265 ==============================
   IntOperation  0.481 =
  LongOperation  0.873 ==
```

Рис. 4. Результаты оценки накладных затрат на вызов JNI функций

Таким образом, возможно предложить решение данной задачи убрав максимальное количество избыточных проверок и встроенных защит нативной имплементации, что позволительно на таком низком уровне выполнения, поскольку предусматривает применение в составе боевых, отлаженных и стабильных системах, где в принципе не допустимо поступление не валидных входных параметров для вычислений, поскольку ответственность за такие проверки и препроцессинг в таких системах несут верхние слои абстракции и реализации системы.

Помимо этого, возможно оптимизировать представленный наивный алгоритм произведения матриц посредством развертывания циклов, введения ряда константных индексов, пакетных буферов матриц, указателей, операций, применения агрессивных уровней оптимизации компилятора нативного решения.

В результате применения предложенных изменений удалось достичь следующих показателей производительности произведения (табл. 3, рис. 4):

Таблица 3

Итерация, №	Затраты процессорного времени, мс
1	2488
2	2382
3	2326
4	2339
5	2292
6	2319
7	2352
8	2317
9	2410
10	2455
Усредненный результат	2368

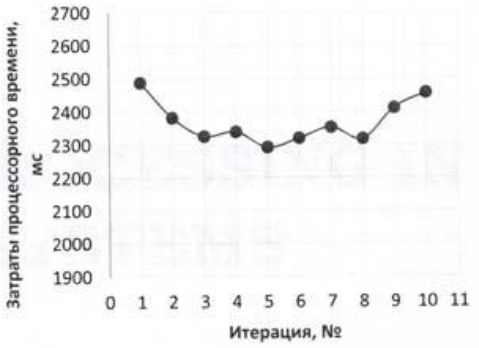


Рис. 4. График зависимости затрат процессорного времени от итерации тестирования для предложенного решения

Выводы

Таким образом, предложенное проприетарное решение показывает результаты в среднем в 1,3 раза лучшие в сравнении с наивным Java-решением и в 2,5 раза лучшие в сравнении с нативным решением из состава SDK.

В перспективе предлагается улучшить предложенное решение посредством реализации полноценного пакетного произведения буферов матриц, максимально сократив таким образом накладные затраты на взаимодействие с нативными функциями.

Литература

1. Article - World, View and Projection Transformation Matrices [Електронный ресурс] – Режим доступу до ресурсу:

http://www.codinglabs.net/article_world_view_projection_matrix.aspx.

2. Don't Get Caught In the Cold, Warm-up Your JVM [Електронный ресурс] / [D. Lion, A. Chiu, H. Sun та ін.]. – 2016. – Режим доступу до ресурсу: http://www.eecg.toronto.edu/~yuan/papers/osdi16-hottub.pdf.

3. Tech Specs [Електронный ресурс] – Режим доступу до ресурсу: https://store.google.com/us/product/pixel_phone_specs?hl=en-US.

.

4. Optimizations [Електронный ресурс] – Режим доступу до ресурсу: https://www.guardsquare.com/en/proguard/manual/optimizations.

5. xref: /frameworks/base/core/jni/android/opengl/util.cpp [Електронный ресурс] – Режим доступу до ресурсу: http://androidxref.com/8.0.0_r4/xref/frameworks/base/core/jni/android/opengl/util.cpp.

6. ProjectHome [Електронный ресурс] – Режим доступу до ресурсу: https://github.com/google/caliper/wiki/ProjectHome